SimCADO Documentation

Release 0.7

Kieran Leschinski and Oliver Czoske

Jul 14, 2022

CONTENTS:

The (slowly expanding) documentation base for SimCADO, the instrument data simulation package for MICADO at the ELT.

Important: SimCADO is currently under active development. The current version of SimCADO (dubbed v0.7) is however capable to answer many questions about the future performance of MICADO imaging mode and your favorite science case. For the spectroscopic mode, please consult SpecCADO.

In this latest version SimCADO is able to correctly simulate DIT and NDITs. Accordingly, the keyword `OBS_EXPTIME` has been replaced by `OBS_DIT`, with total observing time given by `OBS_DIT*OBS_NDIT`. Please update your scripts and configuration files after downloading the latest version.

CHAPTER

ONE

SIMCADO IN A NUTSHELL

SimCADO is a python package designed to simulate the effects of the Atmosphere, E-ELT, and MICADO instrument on incoming light. The current version (v0.7) can simulate the MICADO imaging mode (4mas and 1.5mas per pixel in the wavelength range 0.7µm to 2.5µm).

Fig. 1: A simple simulation in 30 seconds with SimCADO.

1.1 Reference Material

- The inner workings of SimCADO are described in detail in Leschinski et al. (2016)
- The current status of MICADO is described in Davies et al. (2018)

CHAPTER

TWO

DOWNLOADING AND INSTALLING

For more information, see the *Download* section

SimCADO has only been tested in Python 3.x.

It is hightly recommended to use Python 3, however the basics of generating images will still work in Python 2.7. We cannot guarantee this though.

The quick way:

```
$ pip install SimCADO
```

We also recomend to install SimCADO in an anaconda/miniconda/etc python distributions as it is much easier to keep control over your python environment.

The **first time** in python you should update the SimCADO data directory with `get_extras()`:

```
>>> import simcado
>>> simcado.get_extras()
>>>
>>> # !! Only works in Python 3 - See Downloads section
>>> simcado.install_noise_cube()
```

If you running Python 3, it would be helpful to expand the internal detector noise cube with `install_noise_cube()`.

2.1 Keeping SimCADO updated

As MICADO developes, the data files that SimCADO uses will also be updated. Therefore before you do any major work with SimCADO we **HIGHLY** recommend calling `get_extras()`:

```
>>> simcado.get_extras()
```

2.2 iPython/Jupyter notebooks

A (continualy expanding) series of iPython Notebooks detailing how to use SimCADO are available here in the Notebooks section.

Hint: Don't feel like sifting through documentation? Common commands and examples are on the SimCADO cheat-sheet:

- PDF version or
- Jupyter Notebook
- Presentation for October 2017 Science Team Meeting

CHAPTER

THREE

RUNNING A SIMULATION IN 3 LINES

The easiest way to run a simulation is to create, or load, a Source object and then call the `run()` command. If you specify a filename, the resulting image will be output to a FITS file under that name. If you do not specify a filename, the output will be returned to the console/notebook as an `HDUList` object.

To begin, we will import the simcado module (assuming it is already installed).:

```
>>> import simcado
```

At the very least, we need to create a `Source` object which contains both spatial and spectral information on our object of interest. Here we use the built-in command `simcado.source.cluster()` to create a `Source` object for a 10000-Msun stellar cluster. (*Creating Sources* for more information).:

```
>>> src = simcado.source.cluster()
```

We now pass the `Source` object through SimCADO. This is as easy as calling `run()`. If we specify a `filename`, SimCADO will write the output to disk in the form of a FITS file. If no `filename` is given, then SimCADO returns an astropy `fits` object to the console or notebook.:

```
>>> simcado.run(src, filename="my_first_sim.fits")
```

3.1 Changing simulation parameters

The `run()` also takes any *configuration keywords* as parameters for running the simulation. For example, the default exposure time for the simulation is 60 seconds, however this can be increased of decreased by using the keyword *OBS_EXPTIME* (and/or combining it with *OBS_NDIT*). A stacked 6x 10 minute observation sequence would look like:

```
>>> simcado.run(src, filename="my_first_sim.fits", OBS_DIT=600, OBS_NDIT=6)
```

That's it. Of course SimCADO can also go in the other direction, providing many more levels of complexity, but for that the reader is directed to the examples pages and/or the *API documentation*.

CHAPTER

FOUR

SIMCADO BUILDING BLOCKS

For a brief explanation of how SimCADO works and which classes are relevant, please see either the *GettingStarted* or *SimCADO in depth* section.

CHAPTER

FIVE

USING SIMCADO

5.1 Getting Started with SimCADO

SimCADO can be super easy to use, or super complicated. The level of complexity is completely up to the user. A basic simulation involves only 1 thing: a `Source` object to describe the observable object. Once the user has created this object, the function `simcado.simulation.run()` is all that needs to be called. Controlling the parameters of the simulation can be done either by passing keyword-value pairs, or my using a `UserCommands` dictionary.

Contents

Note: Even though this documentation is not yet complete (it is a very big job), lots **more information is included in the docstrings** of every SimCADO function and class. These can be easily viewed in the interactive python interface (iPython, or Jupyter Notebook) with either the question mark operator or by using `SHIFT+TAB` with the cursor over the function name.

Doing this in iPython will call up the docstring:

```
>>> simcado.Source?
```

5.1.1 Source

For full details, please see the *API* and examples of *Source Objects*

The `Source` class is probably the most important class for testing science cases. Therefore spending time on creating accurate `Source` representations of the object of interest is key to getting good results with SimCADO. `Source` objects can be created from scratch, with functions provided by SimCADO, or by loading in a pre-existing `Source`-FITS file.

Note: SimCADO is CaSe SensITIVe! SimCADO has the class `simcado.Source()` and the module *`simcado.source`*. These should not be confused. *`simcado.source`* is the module which contains the class `Source` and all the helper functions for creating various types of `Source` objects. The source code for the class `Source` is actually in `simcado.source.Source`, however to make things easy, `Source` is available directly as `simcado.Source()`. Be careful and remember `simcado.Source != simcado.source`.

For a description of the `Source` object, and the *`source`* module, see How SimCADO works.

Loading a pre-existing `Source` object

To load in a pre-existing `Source` (i.e. one that you saved earlier), specify the keyword `filename=` when initialising the `Source` object.:

```
>>> import simcado as sim
>>> my_src = sim.Source(filename="star_grid.fits")
```

`Source`-FITS files have a very specific file format, so it's best to only import files that were generated directly from other `Source` objects. It's a chicken/egg scenario, which is why the next section deals with creating `Source` objects in memory. For a description of the file format for saved `Source` objects, see "File Format of saved Source objects".

Making a `Source` with SimCADO's in-built functions

The *`simcado.source`* module provides an ever-increasing series of functions to create `Source` objects in memory. These include, (from *`simcado.source`*)

- `empty_sky()`
- `star(mag, filter_name="K", ...)`
- `stars(mags, x, y, ...)`
- `cluster(mass=1E4, distance=50000, ...)`
- `source_from_image(images, lam, spectra, pix_res, ...)`

Two useful functions here are `stars()` and `source_from_image()`

- `stars()` takes a list of magnitudes (and optionally spectral types) and positions for a common broad-band filter (default is "K") and generates a `Source` object with those stars in the field.

```
>>> x, y = [-2.5, 0.7, 16.3], [3.3, -0.2, 25.1]
>>> mags, spec_types = [25,21,28], ["K0V", "A0III", "G2V"]
>>> filt = "H"
>>>
>>> my_src = sim.source.stars(mags=mags, x=x, y=y, filter_name=filt,
                                                spec_types=spec_types)
```

- `source_from_image()` creates a `Source` based on a 2D numpy array provided by the user. The 2D array can come from anywhere, e.g. the data from a FITS image, a BITMAP image, from memory, etc. Alongside the image, the user must provide a spectrum (plus a vector with the bin centres) and the pixel field of view (e.g. 0.004 arcsec for MICADO). SimCADO then extracts all pixels from the image which have values above `flux_threshold` (defualt is 0) and saves these pixel coordinates. The spectrum provided is then connected to these pixel, and scaled by the pixel value.

```
>>> # ... Create an image - a circle with a radius of 20 pixels on a
>>> # ... grid 200 pixel wide
>>> XX = np.array([np.arange(-100,101)]*201)
>>> im = np.sqrt(XX**2 + XX.transpose()**2)
>>> im[im>20] = 0; im[im>0] = 1
>>>
>>> # ... Pull in the spectrum for a G2V star with K=20
>>> lam, spec = simcado.source.SED("G2V", filter_name="K", magnitude=20)
>>>
>>> # ... Make the source object
>>> my_src = sim.source.source_from_image(images=im, lam=lam, spectra=spec, plate_
↪scale=0.004)
```

SimCADO also provides a series of spectra for stars and galaxies, however these are meant as a guide to those who are just starting out. For serious work, the user is encouraged to provide their own spectra. More information on the in-built spectra can be found in the *Source Objects example* section.

5.1.2 Simulating with SimCADO

The quick, the dirty and the ugly

As seen on the *index* page, a simulation can be run using 3 lines of code:

```
>>> import simcado
>>> src = simcado.Source(filename="my_source.fits")
>>> simcado.run(src, filename="my_image.fits")
```

The `run()` function is quite powerful. Many users may find that they don't need anything else to run the simulations they need. The full function call looks like this:

```
simcado.run(src, filename=None,
            mode="wide", detector_layout="small",
            cmds=None, opt_train=None, fpa=None,
            return_internals=False,
            **kwargs)
```

Lets pull this function call apart in order of importance to the simulation:

1. `src`: Obviously the more important aspect is the `Source` object. Without a `Source` these is nothing to observe

2. `filename`: Where to save the output FITS file. If `None` is provided (or the parameter is ignored), the output is returned to the user. This comes in handy if you are working in a Jupyter Notebook and wand to play with the output data immediately. Or if you are scripting with SimCADO and don't want to be slowed down by writing all images to disk
3. Two important parameters here are `mode` and `detector_layout`: These two define the MICADO observing modes.

Currently `mode` can be either `"wide"` (4mas/pixel) or `"zoom"` (1.5mas/pixel).

The `detector_layout` can also be changed to speed up simulations of single objects. For example if the galaxy you're interested in is at z=5, you don't need to read out all 9 MICADO chips for each observation. In fact, a 1024x1024 window at the centre of the middle chip will probably be enough. Therefore SimCADO offers the following "layouts" for the detector - "small", "wide", "full". The default is "small".

- `small` - 1x 1k-detector centred in the FoV
- `centre` - 1x 4k-detector centred in the FoV
- `full` - 9x 4k-detector as described by the keyword `FPA_CHIP_LAYOUT`

1. `cmds, opt_train, fpa` are all parameters that allow you to provide custom built parts of the machinary. Say you have a set of commands saved from a previous simulation run which differ from the default values, then you can use these by passing a `UserCommands` object via the `cmd` parameter. The same goes for passing an custom `OpticalTrain` object to `opt_train` and a custom `Detector` object to `fpa`. For more information see the relevant examples sections - `UserCommands` *examples*, `OpticalTrain` examples, `Detector` examples.
2. `return_internals` allows you to do the opposite of the previous three parameters. If you would like to save the `UserCommands`, `Detector` and/or `OpticalTrain` from your simulation run, the by setting `return_internals=True`, SimCADO will return these along with the simulated imagery. **Note** that this only works if `filename=None`.
3. `**kwargs`: Although `kwargs` is the last parameter, it actually allows you to control every aspect of the simulation. `kwargs` takes any keyword-value pair that exist in the SimCADO configuration file, and so you can control single aspects of the simulation by passing these keyword-value pairs to `run()`. For example, you can increase the exposure time of the image by passing

   ```
   simcado.run(src, ... , OBS_DIT=600, INST_FILTER_TC="J", ...)
   ```

A list of all the available keyword-value pairs can be found in the *Keywords section* .

Alternatively you can dump a copy of the default parameters by calling `simcado.commands.dump_defaults()`.

Changing Filters

The keyword `INST_FILTER_TC` allows you to supply either the name of a filter (i.e. "Ks", "PaBeta") or a path to an ASCII file containing a filter curve. `INST_FILTER_TC` can be passed to `run()` just like any other SimCADO configuration keyword:

```
>>> simcado.run(src, INST_FILTER_TC="J")
>>> simcado.run(src, INST_FILTER_TC="path/to/my_filter_curve.txt")
```

SimCADO has some generic filters built in. These include all the regular NIR broadband filters (I, z, Y, J, H, K, Ks). There are also some narrow band filter. As the MICADO filter set is expected to change, we will not list the SimCADO filter set here. Instead the user can find out which filters are available by calling the function (as of Nov 2016):

```
>>> print(sim.optics.get_filter_set())
['B', 'BrGamma', 'CH4_169', 'CH4_227', 'FeII_166', 'H', 'H2O_204', 'H2_212',
 'Hcont_158', 'I', 'J', 'K', 'Ks', 'NH3_153', 'PaBeta', 'R', 'U', 'V', 'Y',
 'z']
```

If you'd like to use your own filter curve, note that the ASCII file should contain two columns - the first holds the wavelength values and the second hold the transmission values between 0 and 1.

Setting the observation sequence

The important keywords here are: `OBS_DIT`, `OBS_NDIT`

- `OBS_DIT` [in seconds] sets the length of a single exposure. The default setting is for a 60s exposure
- `OBS_NDIT` sets how many exposures are taken. The default is 1.

Depending on what your intended use for SimCADO is, the keyword `OBS_SAVE_ALL_FRAMES=["no", "yes"]` could also be useful. The default is to **not** save all the individual exposures, but stack them and return a single HDU object (or save to a single FITS file). If `OBS_SAVE_ALL_FRAMES="yes"`, then a `filename` must also be given so that each and every DIT can be saved to disk.

Reading out the detector

Warning: running a full simulation could take ~10 minutes, depending on how much RAM you have available:

```
>>> simcado.run(detector_layout="small"")
```

The `detector_layout` keyword is key:

```
detector_layout : str, optional
    ["small", "centre", "full"] Default is "small".
```

Where each of the strings means:

- `"small"` - 1x 1k-detector centred in the FoV
- `"centre"` - 1x 4k-detector centred in the FoV
- `"full"` - 9x 4k-detector as per MICADO imaging mode (either 4mas or 1.5mas)
- `"default"` - depends on "mode" keyword. Full MICADO 9 chip detector array for either 4mas or 1.5mas modes

PSF utilities in SimCADO

In SimCADO we have pre-packaged some simulated PSFs which match the expectations of the different AO modes of MICADO. As new simulations of the AO capabilities become available we will include these new PSFs in SimCADO. The available PSFs are

- PSF_SCAO.fits
- PSF_MCAO.fits

which are the AO modes available. Additionally we provide a LTAO PSF (PSF_LTAO.fits) and a EELT diffraction limited PSF (PSF_POPPY.fits) calculated with poppy

For the moment, the PSF is assumed constant accross the field. The new (refractored) version will be capable to deal with field varying PSFs in a realistic manner.

SimCADO also provides utility functions that are able to produce analytic PSFs to be used in the simulation in adition to the pre-calculated PSFs. Please check the *simcado.psf* module. The most important functions are the following:

- `simcado.psf.poppy_eelt_psf()` creates a diffraction limited PSF based with mirror segments provided by `simcado.psf.get_eelt_segments()`
- `simcado.psf.seeing_psf()` creates a seeing limited PSF with an user provided FWHM. Moffat and Gaussian profiles are available
- `simcado.psf.poppy_ao_psf()` creates an analytical AO PSF with a user provided Strehl ratio

All these functions can save the computed PSFs in fits format by specifying a filename. That file can be later used in the simulations as a parameter in `simcado.simulation.run()` using the `SCOPE_PSF_FILE=filename` keyword.

In the notebook section you can find a few detailed examples how to create and work with these PSFs.

5.1.3 Saving and reusing commands

The `UserCommands` object

Passing more than a few keyword-value pairs to the `simcado.run()` becomes tedious. SimCADO therefore provides a dictionary of commands so that you can keep track of everthing that is happening in a simulation.

```
>>> my_cmds = simcado.UserCommands()
>>> simcado.run(my_src, cmds=my_cmds)
```

When initialised the `UserCommands` object contains all the default values for MICADO, as given in *Keywords*. The `UserCommands` object is used just like a normal python dictionary:

```
>>> my_cmds["OBS_DIT"] = 180
>>> my_cmds["OBS_DIT"]
180.0
```

It can be saved to disk and re-read later on:

```
>>> my_cmds.writeto("path/to/new_cmds.txt")
>>> new_cmds = simcado.UserCommands("path/to/new_cmds.txt")
>>> new_cmds["OBS_DIT"]
180.0
```

If you prefer not to use interactive python and just want to dump a commands file to edit in your favourite text editor:

```
>>> simcado.commands.dump_defaults("path/to/cmds_file.txt")
```

More information on the `UserCommands` object is given in the *Examples Section*

5.1.4 Behind the scenes of SimCADO

SimCADO uses 4 main classes during a simulation:

- `Source` holds spatial and spectral information about the astronomical source of photons, e.g. galaxy, star cluster, etc.
- `OpticalTrain` contains information on the various elements along the optical path, e.g. mirrors reflectivity curves, PSFs, instrumental distortion, etc.

- `Detector` represents the focal plane detector array and contains information on the electronic characteristics of the detector chips and their physical positions.
- `UserCommands` is a dictionary of all the important keywords needed by SimCADO to run the simultationm, e.g. `OBS_DIT` (exposure time) or `INST_FILTER_TC` (filter curve)

For more information on how SimCADO works please see the *SimCADO in Depth* section.

5.1.5 Things to watch out for

This space. It will soon expand!

5.2 Installation

The latest stable version of SimCADO can be installed from PyPI with

```
pip install simcado
```

If you have SimCADO already installed and you want to install the latest version, please type

```
pip install --upgrade simcado
```

Please do not forget to obtain the latest data file package as explained in the section below

The latest development version of SimCADO can be found on GitHub:

https://github.com/astronomyk/SimCADO

5.2.1 Python 3 vs Python 2

Note: SimCADO has been programmed in Python 3.

While most of the basic functionality for SimCADO will work with Python 2.7, we haven't tested it properly. For example, the function `simcado.install_noise_cube()` only works in Python 3. This isn't critical - it just means that if you want to have read noise variations in your images, you need to use Python 3. However it won't crash SimCADO for single images.

Note: A side note: Astropy will stop supporting Python 2.7 in 2019 and the official End-of-Life for Python 2.7 is 2020, i.e. no more maintainance. We are running under the assumption that SimCADO will (hopefully) still be around after 2020, hence why we have concentrated our efforts on developing in Python 3.

Note: SimCADO will need to download several hundreds of MBs of instrument data into the install directory. Hence why we use the *–user* flag when installing via *pip*. If you want to keep SimCADO in your normal packages directory, then you will need to give python root access while updating SimCADO's data files.

5.2.2 Dependencies

Required

Package	Version
numpy	>1.10.4
scipy	>0.17
astropy	>1.1.2
wget	>3.0
requests	>2.0
synphot	>0.1
pyyaml	

Optional

Package	Version
matplotlib	>1.5.0
poppy	>0.4

All dependencies should be automatically installed when typing the *pip* command above. In case of any error message you can try to install them manually:

```
$ pip install numpy scipy astropy wget matplotlib poppy pyyaml synphot requests
```

We really recomend to install SimCADO in a python distribution like anaconda or miniconda as they allow to have full control over your python environment without interfering with your system instalation.

5.2.3 Getting up-to-date data for SimCADO

SimCADO is being developed along side MICADO. To keep the files that SimCADO uses as fresh as possible, you should run the following command, at the very least **the first time you start SimCADO**:

```
>>> import simcado
>>> simcado.get_extras()
```

This downloads the latest versions of all the files SimCADO uses behind the scenes, e.g. PSF files, transmission curves etc. If you haven't used SimCADO for several months, chances are there are new data available. It is therefore advantageous to run this command periodically.

5.2.4 Adding variation to the Detector noise

Note: Currently only available for Python 3.

By default SimCADO only supplies a single H4RG noise frame. Hence if you plan on generating a series of images, all images will have the same *detector noise* pattern. Don't worry, photon shot noise is still completely random, however if you stack 1000s of simulated images with the same detector noise frame, this pattern will show through. This problem can be avoided by running the following function the first time you run simcado:

We recommend running this command in a separate Python session as it can take up to 10 minutes depending on your computer.:

```
>>> simcado.install_noise_cube(n=25)
```

SimCADO contains the code to generate unique detector noise images (Rauscher 2015), however creating a 4k detector noise frame takes about 20 seconds. In order to avoid wasting time by generating noise for each frame every time SimCADO is run, `.install_noise_cube(n)` generates `n` noise frames and saves them in the SimCADO data directory. In future simulation one of these detector noise frames are picked at random whenever a `<Detector>.read_out()` is called.

Obviously a trade-off has to be made when running `.install_noise_cube(n)`. The more noise frames available, the less systematic noise is visible in the read noise of stacked images. However, the more frames are generated, the longer it takes. A good solution is to open a separate window and have SimCADO generate frames in the background.

5.3 SimCADO FAQs

Here are some answers to known issues with SimCADO.

5.3.1 Work around for failing `install_noise_cube()` with Python 2.7

The problem lies with Python 2.7. The noise cube code is 3rd party code that only works on Python 3 and I haven't had a chance to dig into that code yet to find the problem

Generate a noise cube in Python 3. First install python3:

```
$ pip install simcado
$ python3
```

Then create the noise cube:

```
>>> import simcado
>>> sim.detector.make_noise_cube(num_layers=25, filename='FPA_noise.fits',␣
↪multicore=True)
```

Note: Your simcado/data folder can be found by printing the `__pkg_dir__` variable:

```
>>> simcado.utils.__pkg_dir__
```

Copy the new noise cube into the Python 2.7 simcado/data folder.

By default SimCADO looks for the noise cube in its data directory - `<Your Python 2.7 Directory>/lib/python/site-packages/simcado/data/`:

```
$ cp ./FPA_noise.fits  <Your Python 2.7 Directory>/lib/python/site-packages/simcado/
↪data/FPA_noise.fits
```

No access to the `simcado/data` folder?

If you can't save files into the *simcado/data* directory (or you can't be bothered finding it), you can use the FPA_NOISE_PATH keyword when running a simulation to point simcado to the new noise cube file:

```
>>> simcado.run(my_source, ..... , FPA_NOISE_PATH="<path/to/new>/FPA_noise.fits")
```

or if you're using a UserCommands object to control the simulation:

```
>>> cmds = simcado.UserCommands()
>>> cmds["FPA_NOISE_PATH"] = "<path/to/new>/FPA_noise.fits"
```

5.3.2 Surface brightness scaling in SimCADO

Question

Is it true that if the array that I pass to simcado has a value of 1, then simcado will simulate exactly a SB = `m` mag/sq.arcsec pixel? What happens if the passed numpy array has a different pixel size. Are the counts in each pixel then scaled according to their different surface area?

To answer the question on scaling we need look at the docstring for `source_from_image`:

```
source_from_image(images, lam, spectra, plate_scale, oversample=1,
                  units="ph/s/m2", flux_threshold=0,
                  center_pixel_offset=(0, 0),
                  conserve_flux=True)
```

The 3 parameters of interest here are `plate_scale`, `oversample` and `conserve_flux`.

- `plate_scale` is the plate scale of the image. So if we take a HAWK-I image, we have a plate_scale of 0.106 arcsec. SimCADO needs this so that it can work out how many photons are coming in per pixel in the image you provide.
- `oversample`. If oversample stays at 1 (default), then SimCADO will generate a "point source" for each pixel - i.e. every 0.106 arcsec there will be one source emitting with the intensity of that pixel. This is sub-optimal if we want to use an extended object, as SimCADO will turn the image into a grid of point sources with a spacing equal to `plate_scale`. Therefore we need to over sample the image so that SimCADO makes at least 1 light source per pixel. Hence to get down to 4mas for the SimCADO wide field mode, we would need to oversample the HAWK-I image by a factor of 0.106/0.004 = 26.5. I.e. `oversample=26.5`.
- `conserve_flux`. If this is `True` (default), then when SimCADO oversamples the image, it multiplies the new image by a scaling factor `= sum(orig_image) / sum(new_image). Thus a pixel with the value 1 in the original image will now have a value=`(1 / 26.5)^2. `If`conserve_flux=False‘, this scaling factor is not applied. **Note:** The scaling doesn't affect the spectrum associated with the image at all.

5.3.3 Sub-pixel accuracy with SimCADO

There are two ways to go about getting SimCADO to simulate at the sub-pixel level (Nov 2016, only one is working so far):

- Turn on SimCADO's oversample mode with the keyword SIM_OVERSAMPLING:

```
simcado.run(my\_src, ... , SIM\_OVERSAMPLING=4)
```

If SimCADO is using the 0.004 arcsec mode, then it will calculate everything based on a 0.001 arcsec grid. It resamples back up to 0.004 when passing the image to the detector module. Depending on the level of accuracy you need (and the computer power you have available), you can oversample as much as you'd like. Be careful with this - it uses lots of RAM.

- SimCADO also has a sub-pixel mode built into the method `<Source>.apply_optical_train(... , sub_pixel=False)`, - (very slow if there are >100 stars, it doesn't use FFTs) - however I've only done a quick tests to make sure the code works. I can't guarantee it's accurate though. I'll put that on my list of things to do this week (22 Nov 2016)

5.3.4 I have many PSFs in a FITS Cube. How do I use just one layer

To extract a slice from the cube, we use astropy. Here `i` is the layer we want to extract:

```
from astropy.io import fits

f = fits.open("path/to/my/psf_cube.fits")

i = 24      # which ever layer from the cube that you want
psf = f[0].data[i, :,:]
hdr = f[0].header

hdu = fits.PrimaryHDU(data=psf, header=hdr)

hdu.header["CDELT1"] = 0.002     # whatever the plate scale of the PSF file is in␣
↪arcsec
hdu.header["WAVELENG"] = 2.16    # whatever the wavelength of that layer is in micron

hdu.writeto("my_psf_layer.fits")
```

To use this PSF with SimCADO, we use the keyword `SCOPE_PSF_FILE` and pass the filename of the saved PSF slice:

```
simcado.run( ... , SCOPE_PSF_FILE="my_psf_layer.fits", ...)
```

5.3.5 Which filters are included in my version of SimCADO?

SimCADO provides a function to list all the filter curves contained in the *simcado/data* install directory:

```
>>> simcado.optics.get_filter_set()
```

The files containing the spectral response of the curves are located in the *simcado/data* install directory, which can be found by calling:

```
>>> simcado.__data_dir__
```

These files following the naming convention: *TC_filter_<name>.dat*. They contain two columns *Wavelength [um]* and *Transmission [0..1]*

5.3.6 Plotting Filter Transmission curves

To access a transmission curve, use the *get_filter_curve()* function:

```
>>> T_curve = simcado.optics.get_filter_curve(<FilterName>)
```

The returned `TransmissionCurve` object contains two arrays: *.lam* (wavelength) and *.val* (transmission). To access the numpy arrays:

```
>>> wavelength = Tcurve.lam
>>> transmission = Tcurve.val
```

Each `TransmissionCurve` object can be plotted by calling the internal method *.plot()*. The method uses `matplotlib``s current axis (``plt.gca()`), so if you are not using an `iPython` notebook, you will still need to call the `plt.show()` function afterwards. E.g.:

```
>>> import matplotlib.pyplot as plt
>>> T_curve.plot()
>>> plt.show()
```

SimCADO also has a somewhat inflexible function to plot all filter transmission curves which are in the `simcado/data` directory. Basically it loops over all names returned by `get_filter_set()` and plots them. It also applies a nice colour scheme.

```
>>> plot_filter_set()
```

I can also accept a custom list of filter names, if you don't want to plot absolutely everything in the 'simcado/data'directory (fyi, in early versions of simcado this includes many useless files - sorry)

```
>>> plot_filter_set(filters=("J","PaBeta","Ks"),savefig="filters.png")
```

5.3.7 What SimCADO can do?

Many things. Chances are it can do what you'd like, however you may need some patience, and or help from the held desk - see the contact section for who to contact if you have any questions.

5.3.8 What SimCADO can't yet do?

Coronography, Spectroscopy

For the current version of the MICADO spectroscopy simulator see SpecCADO

https://github.com/oczoske/SpecCADO/

We are still working on incorporating SpecCADO into SimCADO, however as the SimCADO <=0.5 was primarily focused on imaging, we need to refactor the core code somewhat in order to achieve this.

5.4 Updates to SimCADO

5.4.1 Data updates

The data files used by SimCADO are continually being updated to reflect the newest information coming from the MICADO work packages.

To update your version of SimCADO, use the `simcado.get_extras()` command.

2019-09-18

- SimCADO can now correctly perform DITs and NDITS
- Accordingly, OBS_EXPTIME keyword in now OBS_DIT

- Documents updated

2019-07-30

- Several examples added to the notebooks
- Bug fixes to source.elliptical

2019-03-26

- SimCADO can now simulate a field varying PSF. See updated docs

2019-03-14

- pip friendly setup.py

2019-02-19

- Documentation Updated
- Reference spectrum now based on synphot

2018-11-23

- Moved the documentation to ReadTheDocs (finally)
- Added filter plotting functionality

2017-01-31

New stable version available: SimCADO 0.4

2017-01-18

- Added E-ELT mirror transmission curve TC_mirror_EELT.dat

2017-01-05

- VLT mirror coating TC_aluminium.dat
- HAWK-I filter curves: TC_filter_J.dat TC_filter_H.dat TC_filter_Ks.dat TC_filter_Y.dat
- Updated detector layout FPA_chip_layout.dat

2016-11-12

- `TC_surface.dat` Updated with more optimistic Strehl values for JHK central wavelengths
- `TC_mirror_gold.dat` Added a reflectivity curve for an unprotected gold coating
- `default.config` Changed `INST_MIRROR_TC` to reference `TC_mirror_gold.dat`

5.4.2 Source Code updates

The current stable version is SimCADO v0.4. The current development version is SimCADO v0.5dev

Development version

2017-01-05 * SimCADO now preloads the transmission curves. Generating an OpticalTrain now only takes ~2 seconds (compared to 20 previously)

2016-12-06 * Added functionality to generate "ideal" AO PSFs using POPPY for the diffraction limited core and added a Seeing halo

2016-11-12 * Bug fix in simcado.psf - psf.lam_bin_centers was taking opt_train.lam_bin_centers instead of using the "WAVE0" keywords in the FITS header * Bug fix in source.psf - apply_optical_train() was asking for PSFs outisde of the opt.train.psf range. related to the point above

5.5 SimCADO Gallery

Images from recent simulations with SimCADO

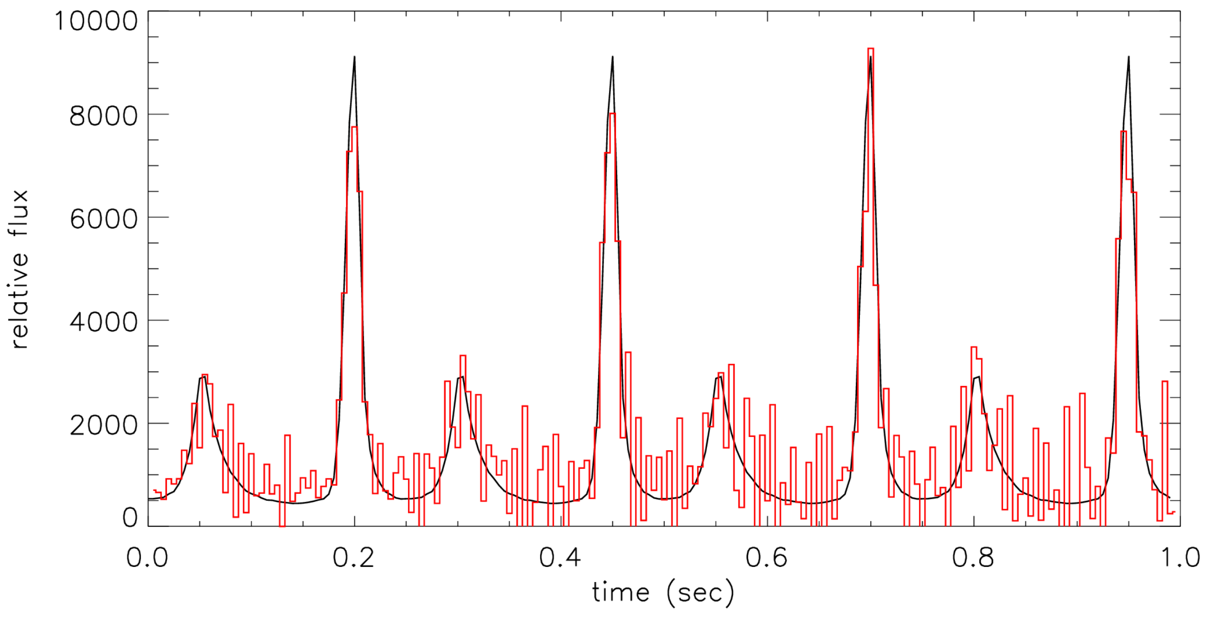


Fig. 1: This is the lightcurve that follows the shape of the Crab Pulsar, but scaled so that each pulse is easily visible in a 5ms windowed H-band exposure with MICADO. The star is H=15mag. The light curve has been extracted from the intermediate non-destructive reads from an up-the-ramp simulated exposure.

5.6 Examples and Tutorials

Here are a list of scripts andnotebooks detailing how to use SimCADO. If you would like to add a notebook to this collection, please email it to Kieran Leschinski.

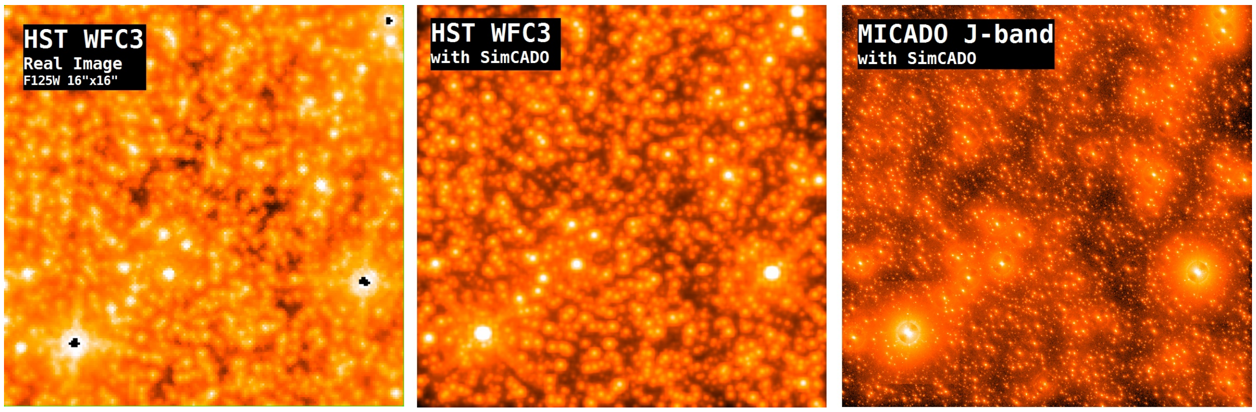


Fig. 2: Omega Cen as imaged with HST/WFC3, HST/SimCADO and MICADO/SimCADO by Maximilian Fabricius (MPE). The synthetic images of the same region of Omega Cen are based on the HST catalog by Anderson & van der Marel 2010 and augmented by all the faint stars that did not end up in the HST catalogue.

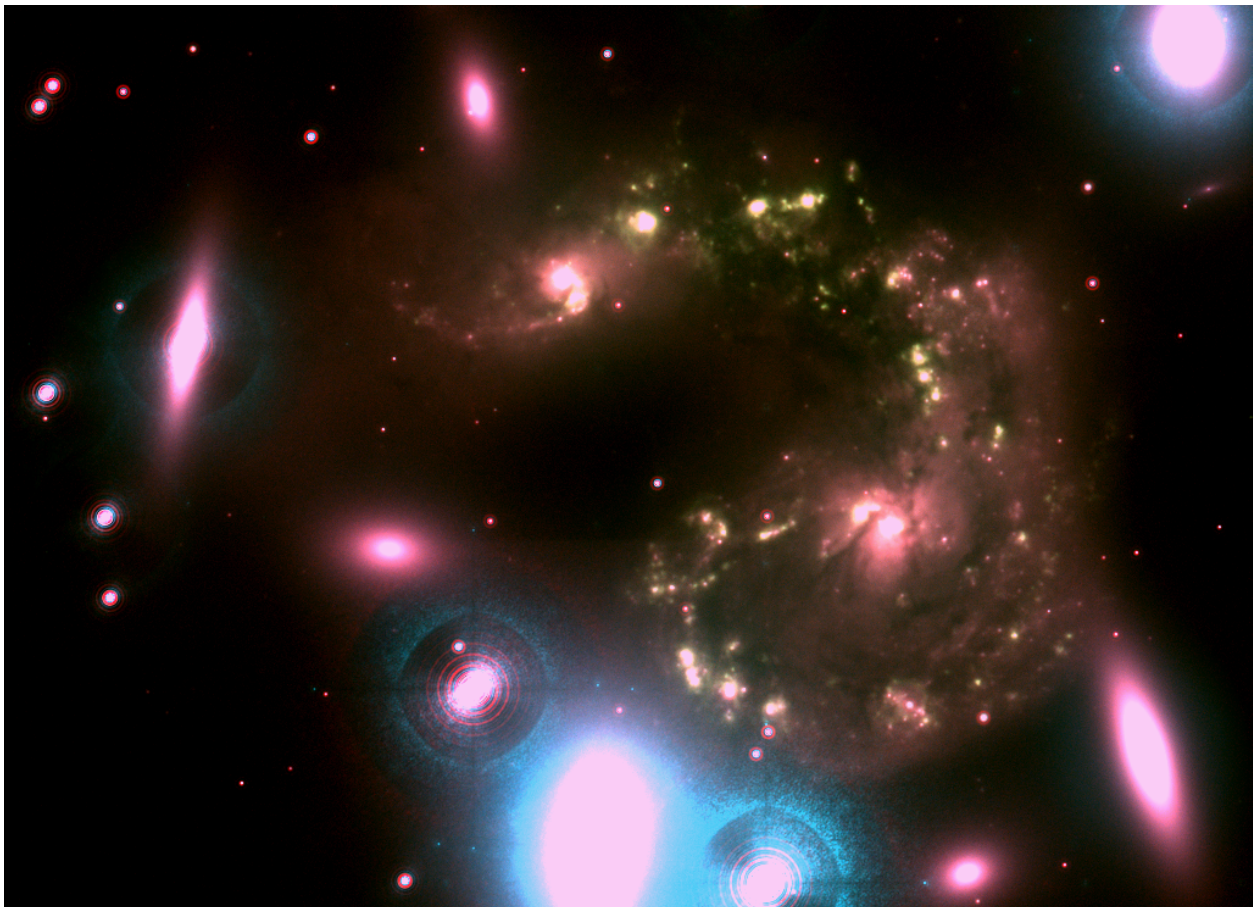


Fig. 3: A galaxy field based on B, R and Halpha images of the Antennae galaxy generated with SimCADO as a proof of concept. The image includes all ways of generating input for SimCADO: importing intensity maps from real (HST) images, generating galaxies based on a sersic profile in memory, and basic stars as point sources with spectra from the Pickles (1998) catalogue. The code to generate this image can be found in the Notebooks section.

5.6.1 Cheat Sheet

Hint: If you don't like sifting through documentation, try looking through the cheat sheet at some common commands and examples. If you find what you need, then you'll know exactly what to look for in the ../reference/simcado

- PDF version or
- Jupyter Notebook

5.6.2 Tutorials

- my_first_sim.ipynb

  An introductory notebook to SimCADO. Topics include: first steps with SimCADO, creating `Source` objects and customising simulations.

- stars_anisocado.ipynb

  A example of how stars looks like with a field changing PSF as expected in the SCAO mode. Here, a bunch of stars are observed at 0, 7, 14 and 21 arcsec off-axis. Also available as a stand-alone script (stars_anisocado.py)

- star_cluster_fullarray.ipynb

  This example simulates a star cluster filling the whole MICADO array with a MAORY-like PSF in the J-band. For that a new J-band PSF was generated with a Strehl of 7%. This PSF can be downloaded from the data directory (as well as others in H and Ks). Also available as stand-alone script (star_cluster_fullarray.py)

- extended_source.ipynb

  We put here a local galaxy and see how it would look like with MICADO at z=2. We use a SCAO PSF. Stand-alone script: extended_source.py

- galaxy_globclusters.ipynb

  We simulate here a disk galaxy surrounded by (proto-) globular clusters and using a number of observing possibilites with SimCADO. Stand-alone script: galaxy_globclusters.py

5.6.3 Science Cases

- Limiting_Magnitudes.ipynb

  Examples for how to use `limiting_mags()` from the sub-module `simcado.simulation`.

- Betelgeuse.ipynb

  A quick look at whether it would be possible to observe Betelgeuse (J = -3 mag) with SimCADO without destroying the detectors. Short answer: yes.

- StarDestroyer.ipynb

  If General Tarkin were to attack Earth and has hidding in orbit around the moon in his spaceship, could the E-ELT see it, and would we we know what it was?

- Examples_Galaxies.ipynb

  A quick look to some of the cababilities of SimCADO to work with extended sources

5.6.4 PSF experiments

- SimCADO_PSF_examples.ipynb

  A notebook showing the different PSFs available in SimCADO and doing a bit of analysis of their characteristics.

- E-ELT_2-Phase_Mirror.ipynb

  As the primary mirror of the E-ELT was innitially thought to be built in 2 phases, this notebook uses SimCADO to have a look at what that means for the resulting E-ELT PSF.

Note: the PSFs generated here are only approximations based on a diffraction limited core combined with a Seeing Halo. These are meant only as a guide.

5.6.5 Workshop notebooks

1. Setting up SimCADO

   Some basics about running SimCADO

2. Working with point sources

   The basics for making and combining point source objects

3. Notes on simulation with sub-pixel resolution

   A couple of examples on how to get SimCADO to do sub-pixel scale simulations

4. Working with extended sources

   Creating a galaxy field with a large galaxy (the Antennae) and adding a series of background galaxies Here are the FITS file used in the simulation

5.7 Using PSFs with SimCADO

Point spread function (PSF) kernels are used by SimCADO to mimic the spread in the beam due to the specific optical configuration of the ELT+MAORY+MICADO optical system.

A PSF kernel is in essence a two dimensional array which describes how the light from a point source is spread out over the detector plane. While SimCADO can accept 2D numpy arrays as input directly from the user, more often than not the user will probably want to use a set of pre-calculated PSFs.

5.7.1 Default PSFs for MICADO

Warning: The default PSFs are approaching their expiry date

The PSF files described below are useful for quick simulations which don't require a super accurate PSF. Furthermore they were generated in 2015 (MCAO, NOVA), and 2016 (SCAO, LESIA), and only cover a single set of atmospheric parameters.

For newer PSFs, see the section on SCAO PSFs (from AnisoCADO) or contact the SimCADO team about MCAO PSFs (from MAROY)

By default the MICADO instrument package includes PSF kernels (arrays) in FITS format for the MCAO and SCAO observing modes. These default kernel FITS files are found in the MICADO data folder under the names: `PSF_MCAO.fits`, `PSF_SCAO.fits`.

SimCADO can be told to use either one of these PSFs by passing the `SCOPE_PSF_FILE` keyword to either the `simcado.run` function, or to a `UserCommands` dictionary:

```
hdu = simcado.run(... , SCOPE_PSF_FILE="PSF_MCAO.fits")
```

or:

```
cmds = simcado.UserCommands()
cmds["SCOPE_PSF_FILE"] = "PSF_MCAO.fits"
```

Note: The default PSFs are field-constant.

This means that they do not vary over the field of view. For the MCAO option this is the optimal case - a constant AO correction everywhere in the field - however for the SCAO case, this is only valid for the central ~2x2 arcsec.

If we are interested in simulating the full MICADO field of view, it will be unrealistic to use the `PSF_SCAO.fits` file. More on this below.

Both the MCAO and SCAO files were generated for reasonably good conditions and should be understood as offering an optimistic view of how MICADO will perform.

5.7.2 Field varying SCAO PSF

As stated above the default PSFs contained in the MICADO package are for the case where the PSF doesn't vary over the field. For the SCAO observation modes this is an unrealistic assumption unless one is only interested in the central ~2x2 arcsec or so. SimCADO (v0.6 and above) accepts a so-called FV-PSF file (field-varying PSF file) in order to simulate the degredation of the PSF over the field of view.

FV-PSF files are quite large (several 100 MB to GB) as they contain PSF kernels for many different positions over the field (and generally for several different wavelengths).

We have generated SCAO FV-PSF files for three observing conditions which roughly correspond to "good", "median", and "bad" atmospheric conditions.

Warning: These files are ~1 GB in size.

Profile	Seeing	Zenith Distance	Wind Speed	Turbulence Profile	File Link
Good	0.4	0	8.8	ESO Quartile 1	AnisoCADO_Q1
Median	0.67	30	10	ESO Median	AnisoCADO_Median
Bad	1.0	60	13	ESO Quartile 4	AnisoCADO_Q4

This FV-PSFs were generated by AnisoCADO. AnisoCADO is python package derived from Eric Gendron's code for generating SCAO PSFs for the ELT+MICADO for any wavelength and off-axis guide star position. To generate custom FV-PSF kernels, see AnisoCADO's documentation for how to make SimCADO-readable FV-PSF FITS files.

Images and profiles of the central PSF from each of these files can be found here:

To use these PSF files we simply pass the filename to SimCADO (as with the other PSF), and SimCADO (! >=v0.6) does the rest:

```
fvpsf_path = "path/to/AnisoCADO_SCAO_FVPSF_4mas_EsoQ1_20190328.fits"
simcado.run(... , SCOPE_PSF_FILE=fvpsf_path)
```

Warning: SimCADO will (probably) run slower when using a FV-PSF file.

This is because SimCADO convolves the FOV of each detector with each PSF kernel that is present inside the borders of that FOV. This means, e.g., if there are 9 different PSFs to be used in the region covered by the central detector chip, then this region will be convolved 9 times with different PSFs (and masked accordingly 9 times). Hence simulating the central chip will take 9 times as long as when using a field-constant PSF.

5.8 Custom instrument packages for SimCADO

Warning: This format will only work with SimCADO v0.4 and later versions

This way of specifying files and parameters has been depreciated for newer versions of SimCADO. A new API description will be available by June 2019.

SimCADO is a (reasonably) flexible framework for simulating telescope and instrument optical trains. It was primarily developed for use with the ELT telescope and the MICADO / MAORY instrument, however many other instruments can be simulated with SimCADO if the proper configuration files are provided.

This page aims to give a short **overview of which files** SimCADO needs in order to simulate a custom optical train.

Note: We're happy to help implement other instruments with SimCADO

If you would like to use SimCADO to model another telescope / instrument system, and would like some help with how to do this, please contact the SimCADO team. We're more that happy to help out.

5.8.1 Config Files

The default SimCADO configuration file can be found in the data folder of the installation directory: `simcado/data/default.config`. This folder can be found by calling:

```
import simcado
simcado.__data_dir__
```

To create a custom instrument package, we will need to create our own `.config` file. We can dump the default file by calling:

```
simcado.commands.dump_defaults("my_new_instrument.config", selection="all")
```

Next we will need to alter the values to fit our instrument. For some keywords we will need to provide file paths to a (correctly formatted) file describing the given effect. The most important files are described below.

5.8.2 File paths

Note: Use `simcado.__search_path__` to store files in multiple places

SimCADO contains a function (`find_file`) which looks for file names in the directories contained in the list `simcado.__search_path__`. By default this list contains the working directory, and the two main folders in the installation directory:

```
>>> simcado.__search_path__
['./', '<pkg_dir>/simcado', '<pkg_dir>/simcado/data']
```

By adding directory names to the `__search_path__` list at the beginning of a python script/session, we can tell SimCADO to look in other directories for the relevant files.

5.8.3 Required Files

PSF files

Keywords : SCOPE_PSF_FILE

This file should contain a (series of) PSF kernel(s) as FITS image extensions. Each extension must contain the keyword `WAVE0` which tells SimCADO the wavelength for which the PSF kernel is relevant (in `m` or in `um`).

For MICADO, the default PSF FITS file contains extensions for each of the major broadband filters: I [0.87um], z [1.05um], J [1.25um], H [1.65um], Ks [2.15um]. For filters with central wavelengths which fall between these PSF kernels, SimCADO takes the extension with the closest wavelength.

In case we need to include field-variations in the PSF cube, the new PSF format is also accepted by SimCADO. It's description can be found here :

Mirror transmission curves

Keywords : SCOPE_M1_TC, INST_MIRROR_AO_TC, INST_MIRROR_TC, INST_FILTER_TC

TC (transmission curve) files are ASCII table files with two columns. The transmission profile can be as simple or as detailed as we want. Here is an example of a very simple filter transmission file:

```
# TC_filter_Ks.dat
Wavelength  Transmission
0.1         0
1.89        0
1.9         1
2.4         1
2.41        0
3.0         0
```

SimCADO uses `astropy.table` to read the file, so if `astropy` can read it, SimCADO can too.

Mirror lists

Keywords : SCOPE_MIRROR_LIST, INST_MIRROR_AO_LIST

These list files contain information on the geometry of the mirrors included in either the telescope section, or AO section of the optical system. Below we have the first 3 lines of the ELT's mirror list file:

```
# EC_mirrors_scope.tbl
Mirror      Outer    Inner    Angle   Temp     Coating
M1          37.3     11.1     0.      0.       TC_mirror_mgf2agal.dat
```

where Outer, Inner [m] refer to the diameter of the mirror, Angle [deg] is the angle at which the plane of the mirror differs from that of the light beam, Temp [deg C] is the mirror temperature for determining the thermal background, and Coating is the filename of the Transmission Curve (TC) file describing the spectral response of the mirror coating material (see above: Mirror transmission curves).

The internal mirror configuration of the actual instrument is a special case. It is assumed that the cryostat temperature is sufficiently low that the blackbody emission from the internal mirrors is negligible. Hence only the mirror transmission is important and can therefore be described with the following two parameters:

```
INST_NUM_MIRRORS
INST_MIRROR_TC
```

For instruments without AO module, we can force SimCADO to ignore the AO components by setting the following keyword:

```
INST_USE_AO_MIRROR_BG = False
```

Wavefront errors

Keywords : INST_WFE

The current version of SimCADO (v0.6) includes only the reduction in strehl ratio induced by wave front error. It does this by calculating the peak of a normalised 2D Gaussian kernel (i.e. between 1 and 0) for each wavelength in the final system transmission curve, and applying an effective transmission loss based on the Gaussian peak value. This is very much a 'first order approximation' to including wavefront errors in the optical train. It should be noted that this has been improved upon in SimCADO v1.0.

The file passed to `FPA_QE` should follow this table format (taken from the MICADO+ELT `INST_wfe.tbl` file):

```
# wfe_rms     no_surfaces      material     optics
# [nm]        [#]              [str]        [str]
20            11               gold         mirror
10            4                glass        entrance_window
10            2                glass        filter
10            8                glass        ADC
```

5.8.4 Optional files

Atmospheric spectra

Keywords : ATMO_TC, ATMO_EC

By default SimCADO uses precalculated output from the ESO skycalc tool for the atmospheric emission and transmission curves. The default tables (`EC_sky_25.tbl` and `TC_sky_25.tbl`) are for PWV = 2.5mm and include columns for various airmasses over a wavelength range or [0.3, 3.0]um.

Additional transmission curves

Keywords : INST_DICHROIC_TC, INST_ENTR_WINDOW_TC, INST_PUPIL_TC

In case the transmission properties of any dichroics, entrance windows or pupil optics need to be included. The file format is the same as above for *Mirror transmission curves*

Detector noise images

Keywords : FPA_NOISE_PATH

By default SimCADO uses a FITS file containing an image of the detector noise pattern for the HAWAII-4RG detectors. We can include pre-calculated noise maps by passing a FITS file to `FPA_NOISE_PATH`. No special header keywords are needed.

Detector linearity curve

Keywords : FPA_LINEARITY_CURVE

Detector linearity can be included by passing an ASCII table with two columns which relate the real incoming flux to the measured photon flux as measured by the read-out electronics. The table should look like this:

```
# real_flux measured_flux
0           0
1           1
1000        998
3500        3200
...         ...
200000      180000
1000000     180000
```

Note: Linearity is applied to any imaging observation, regardless of length

Yes, this shouldn't be, but we haven't got around to fixing that yet. Hence to model long exposure observations (i.e. >1 min), it's best just to set `FPA_LINEARITY_CURVE = False`

5.9 Simulation Building Blocks

5.9.1 SimCADO objects

Source

The `Source` class is probably the most important class for testing science cases. Therefore spending time on creating accurate `Source` representations of the object of interest is key to getting good results with SimCADO.

Basically a `Source` object represents photon sources using lists of positions (`.x, .y`), a list of unique spectra (`.spectra`) and a list of references which match each photon source to a spectrum in the list of spectra (`.ref`). All sources (extended and point source) can be decomposed into these lists. The advantage of using this approach is that objects with highly similar spectra can both reference the same position in `.spectra`, thereby reducing the number of spectra that need to be manipulated during a simulation.

File Format of saved `Source` objects

`Source` objects are stored as FITS files with 2 extensions. The first (ext=0) contains an image with dimensions (n, 4) where n is the number of positions for which there exists a spectrum - i.e. non-empty space. The 4 rows of the image correspond to the `.x, .y, .ref, .weight` arrays. The second extension (ext=1) contains another image with dimensions (m+1, len(lam)), where m is the number of unique spectra and len(lam) is the length of the `.lam` array containing the centres of all wavelength bins. The first row in the second extension is the `.lam` array.

Todo: describe how `Source` is linked to `OpticalTrain`, `Detector` and `UserCommands`

OpticalTrain

Todo: describe how `OpticalTrain` is linked to `Source`, `Detector` and `UserCommands`

Detector

Todo: describe how `Detector` is linked to `Source`, `OpticalTrain` and `UserCommands`

UserCommands

Todo: descript how `UserCommands` is linked to `Source`, `OpticalTrain` and `Detector`

5.9.2 SimCADO methods

5.9.3 Method behind applying an `OpticalTrain` to a `Source` object

(Taken from Leschinski et al. 2016)

In an ideal world, SimCADO would apply all spectral and spatial changes at the resolution of the input data. However, the memory requirements to do this are well outside the limits of a personal computer. The solution to this problem is to split the effects based on dimensionality. For certain elements in the optical train, the spectral and spatial effects can be decoupled (e.g. purely transmissive elements like the filters versus purely spatial effects like telescope vibration). For other elements, most notably the PSF and ADC, all three dimensions must be considered simultaneously. When applying an `OpticalTrain`, SimCADO follows the procedure described graphically in Figure 1. The example used in Figure 1 is for a simplified stellar cluster with only two different stellar types - A0V and K5V type stars.

5.9.4 Method behind reading out a `Detector` object

Todo: add in a description of how `Detectoctor.read_out()` works

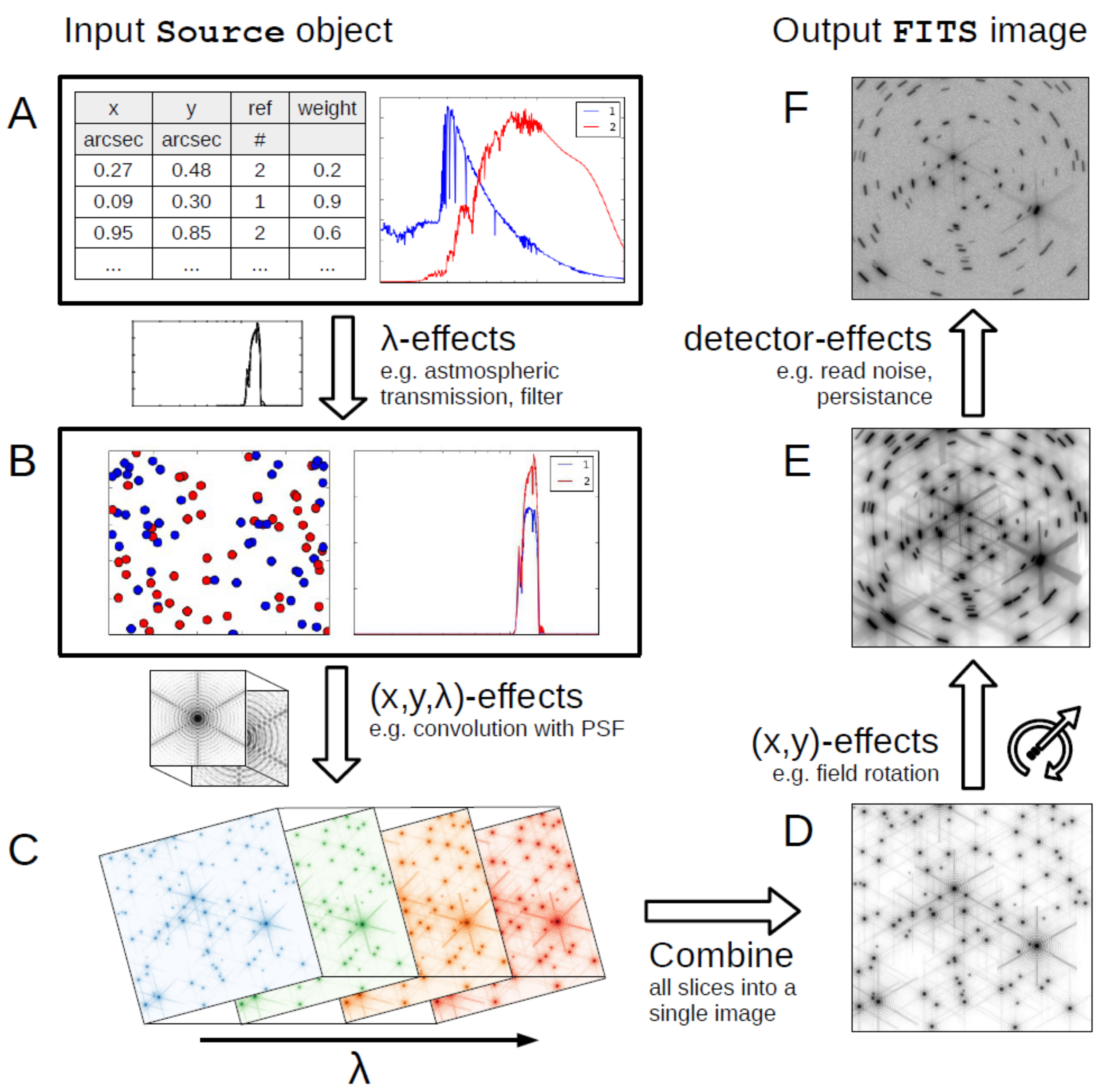


Fig. 4: Figure 1 - The steps involved in applying an `OpticalTrain` object to a `Source` object. The example used here is for a simplified stellar cluster with only two different stellar types - A0V and K5V type stars.

- **A.** The original `Source` object includes an array of spectra for each **unique** photon source (in the case of Figure 1, there are only two unique spectra) and four vectors: `x`, `y`, `ref`, `weight`. `x` and `y` hold the spatial information for each photon source, `ref` connects each source to a spectrum in the array of spectra and `weight` allows the spectrum to be scaled.
- **Spectral-effects**. The first step in `apply_optical_train()` is to combine all optical elements which only act in the wavelength domain (e.g. filters, mirrors, etc.) into a single effect, then apply that effect to the array of spectra in the `Source` object.
- **B.** The spectra in the `Source` object are now representative of the photo-electron count at the detector, assuming a perfect optical train and at the internal spatial resolution of the simulation, i.e. *not at the pixel scale of the detector*. The position vectors are converted into a two-dimensional "image" of the `Source`.
- **Spectrospatial-effects.** The second step includes creating "slices" through the data. The spectra are binned according to several criteria (ADC shift, PSF FWHM difference, etc) with a spectral resolution anywhere from R=1 to R>100, and the number of photons per source in each wavelength bin is calculated. The sources in each "slice" are scaled according to the number of photons in each bin. The relevant spatial effects (atmospheric dispersion, convolution with PSF kernel, etc.) are then applied to each slice in turn.

- **C.** At this stage, the `Source` object contains many spectral slices. Each is essentially the equivalent of a (very) narrow filter image.
- **D.** All spectral effects have been taken into account, and so the binning in the spectral domain is no longer needed. The third step in `apply_optical_train()` is to add all the slices together to create a single monochrome image.

5.10 Controlling simulations

This page describes two ways of simulating: the quick and the repeatable. More about how SimCADO deals with commands can be found in the section on the UserCommands object.

5.10.1 Simulating the quick way

The function `run()` runs a simulation with all the default parameters. That said, SimCADO has a safety switch to stop you wasting 10 minutes on a `Source` that will be invisible, or so bright that it saturates the detector completely.

To get the full detector array, we set `detector_array="full"`:

```
>>> im = sim.run(src, detector_array="full")
```

Note: Note, by patient! Simulating 9x 4k detectors is heavy lifting. Remember to use the internals if you don't plan on changing the optical train.

Notes on simcado.run()

There are two parameters still to be covered in detail. Please see the documention (or read the docstring by using `sim.run?`). They are:

- `filename=` : instead of returning the FITS object to the console, save it to disk under this name
- `mode=` : default is "wide" for 4mas imaging. Also accepts "zoom" for 1.5mas imaging mode.

Viewing a full detector read out

Each chip read-out is stored in a FITS extension. There are 3 options to view the images:

1. Save the FITS object to disk and use DS9, etc:

```
>>> im.writeto("full_detector.fits")
```

2. Use the `filename=` in `simcado.run()` to save directly to disk, and bypass the console. This is useful for scripting with SimCADO.

```
>>> simcado.run(src, filename="full_detector.fits")
```

3. Use the SimCADO function `.plot_detector()` from the `.detector` module:

```
>>> im, (cmd, opt, fpa) = simcado.run(src, filename="full_detector.fits",
                                       return_internals=True)
>>> simcado.detector.plot_detector(fpa)
```

The 3rd option is probably the least favourable as there are no options available, but it allows you to see what the readout will look like in a mosaic mode.

Using KEYWORD=VALUE pairs from the configuration file

SimCADO is controlled with a series of keyword-value pairs contained in a configuration file. **The defaults are the best approximation to MICADO** so changing them is *not* recommended if you want to simulate *MICADO* images. There are however some which are useful to play around with.

Note: SimCADO is CAsE sEnsiTIve**. All KEYWORDS are writen with capital letters.

Similar to SExtractor, SimCADO provides a way to dump both commonly used and less-common keywords to a file with command `sim.commands.dump_defaults()`:

```
>>> sim.commands.dump_defaults()
##########################################################
##             FREQUENTLY USED KEYWORDS                 ##
##########################################################

#--------------------------------------------------------#
# To dump a file with all keywords, use:                 #
# >>>  sim.commands.dump_defaults(filename, type="all") #
#--------------------------------------------------------#

OBS_DIT                 60           # [sec] simulated exposure time
OBS_NDIT                1            # [#] number of exposures taken
INST_FILTER_TC          Ks           # [<filename>, string(filter name)] List␣
↪acceptable filters with >>> simcado.optics.get_filter_set()

ATMO_USE_ATMO_BG        yes          # [yes/no]
SCOPE_USE_MIRROR_BG     yes          # [yes/no]
INST_USE_AO_MIRROR_BG   yes          # [yes/no]
FPA_USE_NOISE           yes          # [yes/no]

FPA_CHIP_LAYOUT         small        # [small/centre/full] description of the chip␣
↪layout on the detector array.
SCOPE_PSF_FILE          scao         # ["scao" (default), <filename>, "ltao", "mcao",
↪"poppy"] import a PSF from a file. Default is <pkg_dir>/data/PSF_SCAO.fits

SIM_DETECTOR_PIX_SCALE  0.004        # [arcsec] plate scale of the detector
SIM_VERBOSE             yes          # [yes/no] print information on the simulation run

OBS_ZENITH_DIST         60           # [deg] from zenith
INST_ADC_PERFORMANCE    100          # [%] how well the ADC does its job
```

To list all keyword-value pairs, use:

```
>>>  sim.commands.dump_defaults(filename, type="all")
```

Any of the KEYWORD=VALUE pairs can be passed as extras to *:func:*.run‘. For example if we wanted to observe in J-band for 60 minutes, we would pass:

```
src = sim.source.source_1E4_Msun_cluster()
im = sim.run(src, OBS_EXPTIME=3600, INST_FILTER_TC="J")
```

The jupyter notebook my_first_sim.ipynb has more exmples of this.

5.10.2 The UserCommands object

Behind the scenes of the `simcado.run()` command, three objects are created:

- a `UserCommands` object - for holding all the information on how a simulation should be run
- an `OpticalTrain` object - which contains the models to describe each effect that needs to be simulated
- a `Detector` object - commonly referred to as an `fpa` or Focal Plane Array. It describes the layout of the detectros and holds the observed images.

The `UserCommands` object is arguably the most important of these three, because the other two need the keyword-value pairs contained within the `UserCommands` object to correctly describe the optical train and detector for the simulation.

A `UserCommands` object is created by reading in the defaults conifg file (`defaults.config`) and then updating any of the keywords that the user (or function) provides. For example, we can see all the default keyword-value pairs by calling:

```
>>> cmd = sim.UserCommands()
```

The `UserCommands` object contains 7 ordered dictionaries, one for each topic and one general dictionary. Each can be referenced individually, however all are updated when a value changes.

1. cmd.cmds - contains all keyword-value pairs
2. cmd.atmo - keyword-value pairs for the atmosphere
3. cmd.scope - keyword-value pairs for the telescope
4. cmd.inst - keyword-value pairs for the instrument (plus AO system)
5. cmd.fpa - keyword-value pairs for the dector array
6. cmd.obs - keyword-value pairs for the observation
7. cmd.sim - keyword-value pairs for the simulation

A `UserCommands` object can be used as a dictionary itself, although technically all that happens is that it references the general dictionary `cmd.cmds`. For example

```
>>> cmd["OBS_DIT"] = 60
```

is exactly the same as either of the following two expressions

```
>>> cmd.cmds["OBS_DIT"] = 60
>>> cmd.obs["OBS_DIT"] = 60
```

Therefore for the sake of ease, we recommoned treating the `UserCommands` object as a dictionary and just using the default syntax: `cmd["..."] = xxx`

Saving and loading a `UserCommands` object

Saving

In case you have made changes to the values in a `UserCommands` object that you would like to keep for next time, a `UserCommands` object can be saved to disk with the following command:

```
>>> cmd = sim.UserCommands()
>>> cmd.writeto(filename="my_cmds.txt")
```

SimCADO writes out the dictionary in ASCII format.

Loading

Creating a `UserCommands` object based on a text file is as simple as passing the file path:

```
>>> cmd = sim.UserCommands("my_cmds.txt")
```

Special attributes

A `UserCommands` object not only contains a dictionary of keyword-value pairs, but also a select number of parameters pertaining to the optical train for quick access. These include values for:

- the exposure time for simulations: `cmd.exptime`
- the primary mirror: `cmd.area`, `cmd.diameter`
- the wavelength vector for purely spectral data (i.e. transmission curves): `cmd.lam`
- the wavelength centres and edges for each spectral bin: `cmd.lam_bin_centres`, `cmd.lam_bin_edges`
- the mirror configuration: `cmd.mirrors_telescope`, `cmd.mirrors_ao`
- the detector plate scale and internal sampling resolutions: `cmd.fpa_res`, `cmd.pix_res`

5.10.3 Mirror and Detector configuration files

A quick note on the other files that SimCADO uses when creating an optical train and the appropriate keywords

The detector array

The detector array is described by a text file containing information on the plate scale and the positions of the detector chips:

```
>>> sim.commands.dump_chip_layout(path=None)
#  id     x_cen     y_cen   x_len   y_len
#         arcsec    arcsec   pixel   pixel
    4         0         0    4096    4096
    0   -17.084   -17.084    4096    4096
    1         0   -17.084    4096    4096
    2    17.084   -17.084    4096    4096
    3   -21.484         0    4096    4096
    5    17.084         0    4096    4096
    6   -17.084    17.084    4096    4096
    7         0    17.084    4096    4096
    8    17.084    17.084    4096    4096
```

This small file can be saved to disk by passing a filename to the `path=` parameters

```
>>> sim.commands.dump_chip_layout(path="my_fpa.txt")
```

Any detector array can be provided to SimCADO, as long as the text file follows this format. For example the HAWK-I detector array (4x HAWAII-2RG) would look like this:

```
#  id     x_cen     y_cen     x_len    y_len
#         arcsec    arcsec    pixel    pixel
    0       -116      -116     2048     2048
    1        116      -116     2048     2048
    2       -116       116     2048     2048
    3        116       116     2048     2048
```

To pass a detector array description to SimCADO, use the `FPA_CHIP_LAYOUT` keyword:

```
>>> cmd = sim.UserCommands()
>>> cmd["FPA_CHIP_LAYOUT"] = "hawki_chip_layout.txt"
```

or pass is directly to the *:func:*.run‘ command:

```
>>> sim.run(... , FPA_CHIP_LAYOUT="hawki_chip_layout.txt", ...)
```

The mirror configurations

The mirror configuration can be dumped either to the screen or to disk by using:

```
>>> dump_mirror_config(path=None, what="scope")
#Mirror      Outer    Inner    Temp
M1           37.3     11.1     0.
M2           4.2      0.545    0.
M3           3.8      0.14     0.
M4           2.4      0.       0.
M5           2.4      0.       0.
```

If `path=None` the contents of the default file are printed to the screen. The parameter `what` is for the section of the optical train that should be shown - either `scope` for the telescope, or `ao` of the AO system. For most existing telescope, this parameter is irrelevant. For the MICADO/MAORY setup however another six optical surfaces are introduced into the system.

It is possible to specifiy different mirror configurations using a text file with the same format as above. For example the VLT unit telescope mirror config files would look like this:

```
#Mirror Outer    Inner    Temp
M1      8.2      1.0      0.
M2      1.116    0.05     0.
M3      1.0      0.       0.
```

To use this mirro config file in SimCADO use the keywords `SCOPE_MIRROR_LIST` and `INST_MIRROR_AO_LIST`

```
>>> cmd = sim.UserCommands()
>>> cmd["SCOPE_MIRROR_LIST"] = "vlt_mirrors.txt"
>>> cmd["INST_MIRROR_AO_LIST"] = "none"
```

5.11 Examples with the Source object

The `Source` class is probably the most important class for testing science cases. Therefore spending time on creating accurate `Source` representations of the object of interest is key to getting good results with SimCADO.

Basically a `Source` object represents photon sources using lists of positions (`.x, .y`), a list of unique spectra (`.spectra`) and a list of references which match each photon source to a spectrum in the list of spectra (`.ref`). All sources (extended and point source) can be decomposed into these lists. The advantage of using this approach is that objects with highly similar spectra can both reference the same position in `.spectra`, thereby reducing the number of spectra that need to be manipulated during a simulation.

Contents

5.11.1 My first Source object

To begin with it is probably easiest to let SimCADO generate a `Source` object. The convenience function `simcado.source.star()` will generate a `Source` object containing a single star. In this case, we'll choose an G2V star with a K-band magnitude of 20, placed 5 arcsec above the centre of the focal plane:

```
>>> star_1 = simcado.source.star(mag=20, filter_name="K", spec_type="G2V", x=5, y=0))
```

`star_1` the coordinates of the star are held in the arrays `.x` and `.y`, and the G2V spectrum in '.spectra´:

```
>>> star_1.x[0], star_1.y[0]
(5, 0)
```

The spectrum for `star_1` is held in `.spectra` and the central wavelength of each of the spectral bins is in `.lam`.

```
>>> star_1.lam, star_1.spectra[0]
<insert output here>
```

Note: *.lam* is a (1,n) array where as *.spectra* is a (m,n) array where n is the number of bins in the spectra and m is the number of unique spectra in the *Source* object. If the *Source* only contains a single unique spectrum, then *.spectra* will be a (1,n) array too.

Combining `Source` objects

Because a `Source` object is just a collection of arrays, it is easy to add many together with the + operator:

```
>>> star_2 = simcado.source.star(mag=22, filter_name="K", spec_type="A0V", x=2, y=-2)
>>> two_stars = star_1 + star_2
>>> two_stars.x, two_stars.y
((5, 2), (0, -2))
```

Note: this is a very trivial example (for which SimCADO has a more elegant function: *simcado.source.stars()*), but it serves to illustrate the main idea. The overloaded + operator is very useful for combining objects, e.g. forground stars and background galaxies, in order to get a better representation of the sky.

Note: if you are planning on creating sources for large numbers of stars (e.g. >>10 stars), using the plural function `stars()` will save you a lot of time.

Point sources

For generating a field of stars, SimCADO offers a series of convenience functions. Please see the docstring or API documentation for more information on how best to use them.

- `simcado.source.star()`
- `simcado.source.stars()`
- `simcado.source.star_grid()`
- `simcado.source.cluster()`

SimCADO's in-built example spectra

Add a section on this

- Stellar templates from Pickles (1998)
  - `simcado.source.SED()`
  - `simcado.source.empty_sky()`
- Galaxy templates from STSCI

Each of these functions returns two arrays: `lam` and `spec`

Using an image as a template for a `Source` object

If we have an extended source that we wish to simulate, e.g. a galaxy, a nebula, etc. we can use the function `simcado.source.source_from_image()`. The image must be a 2D `numpy.ndarray`, but it can come from anywhere, e.g. a FITS file, generated my another function, or even a MS Paint bitmap image.

```
>>> image_1 = astropy.io.fits.getdata("orion.fits")
>>> image_1
array([[0.0, ... 0.0],
       ... ,
       [0.0, ... 0.0]])
```

Here SimCADO takes a the pixel coordinates of the image and converts them to positions on the focal plane.

Note: the user must specify a plate-scale in arcseconds (*pix_res=*) for the image. Each pixel with a value above a certain threshold (default *flux_threshold=0*) will be used in the *Source* object. The coordinates of these pixels are added to the arrays *.x* and *.y*.

We also need to provide a spectrum for the image. This spectrum is assumed to be the only spectrum for each pixel in the image. The pixel values are then the intensity assigned to that spectrum at that pixel position.

SimCADO provides the pickles library for stellar spectra. Unfortunately there aren't any built-in galactic spectra yet - for this the user will need to provide their own spectrum.

```
>>> lam, spec_1 = simcado.source.SED("G2V", "K", magnitude=20)
>>> lam, spec_1
(array([0.7 ... 2.5]), array([0.0 ... 0.0]))
```

With `image_1`, `lam` and `spec_1` we can now build a `Source` object for an orion-like nebula that has the spectrum of a sun-like star.

```
>>> simcado.source.source_from_image(image_1, lam, spec_1, pix_res=0.004, flux_
↪threshold=0)
```

While this example is physically unrealistic, it serves the purpose of showing how to build a `Source` object from an image. The user is

Images with multipe spectra

In reality assigning a single spectrum to an extended object is of limited use. For a `Source` to be realistic is should contain multiple spectra for objects in different locations. The best way to simulate this with SimCADO is to create a `Source` object for each unique group of objects (e.g. old stellar population, star forming regions, AGN, etc) and then combine them into a single `Source` object with the + operator.

As a worked example, lets create a "first-order" approximation to a star forming galaxy. The two major components of this source are 1. the aged stellar population and, 2. the star forming regions.

In our (very) crude model the aged stellar population can be approximated by an ellipse with Gaussian light distribution. As M stars make up the majority of this population, we can assign a M0V spectrum to this population.

```
>>> from astropy.convolution import Gaussian2DKernel
>>> from simcado.source import SED
>>>
>>> old_pop = Gaussian2DKernel(128).array[::3,:]
>>> m0v_spec = SED()
```

To illustrate (very crudely) the star forming regions we can create a random distribution of elliptical Gaussians using the `astropy` function `Gasussian2DKernel`:

Creating a `Source` object from scratch

To create a `Source` object from scratch, we initialise the object by passing 5 (or 6) arrays. All the parameter names must be specified.

`sim.Source(lam=, spectra=, x=, y=, ref=, [weight=])`

where: + `x, y` - [each a `numpy.ndarray`]. Coordinates for each point source in the image in units of [arcsec] from the focal plane centre

- `lam` - [`numpy.ndarray`]. An array with the centre of the wavelength bins in [um] for each unique spectrum
- `spectra` - [`numpy.ndarray`]. An (n, m) array holding n spectra, each with m values. Default units are [ph/s] Note - `lam` and `spectra` should use a constant bin width. Variable bin widths leads to unpredictable results.
- `ref` - [`numpy.ndarray`]. An array to connect the point source at `x[i]`, `y[i]` to a unique spectrum at `spectra[j]`, i.e. `ref[i] = j`

Optional keywords can be specified:

- `weight` - [`numpy.ndarray`], optional. If two sources share the same spectrum, but are at different distances or have different luminosities a scaling factor can be specified to the spectrum when applied to each specific point source.
- `units` [default `"ph/s"`] is the units for the spectra, i.e. n phontons per second per spectral bin. The size of the spectral bins is resolution of the `.lam` array.

Combining two (or more) `Source` objects

`Source` objects can be created in different ways, but the underlying table-structure is the same. Therefore adding `Source` objects together means simply combining tables. The mathematical operator + can be used to do this:

```
>>> # ... create a A0V star at (0,0) and a G2V star at (5,-5)
>>> star_A0V = sim.source.star(20, spec_type="A0V", x=0, y=0)
>>> star_G2V = sim.source.star(20, spec_type="G2V", x=5, y=-5)
>>>
>>> src_combi = star_A0V + star_G2V
>>>
>>> print(src_combi.x, src_combi.y)
[0 5] [ 0 -5]
```

By adding different `Source` objects together, it is possible to build up complex objects that will be representative of the observed sky, e.g. old + new galaxy stellar population + gas emission + foreground stars

See examples for how to use the `*` and − operators with a `Source` object

Saving a `Source` object to disk

The `Source` object is saved as a FITS file with two extensions. See How SimCADO works for more on the file structure.

```
>>> src_combi.write("my_src.fits")
```

The file can be read in at a later time by specifying `filename=` when initialising a `Source` object - as stated above

```
>>> my_src = sim.Source(filename="my_src.fits")
```

In-built `Source` object for a star cluster

As a test object, SimCADO provides the function, with all distances in parsecs

```
sim.source.cluster(mass=1E4, distance=50000, half_light_radius=1)
```

5.11.2 SimCADO convenience functions

- `simcado.source.empty_sky()`
- `simcado.source.stars()`
- `simcado.source.cluster()`
- `simcado.source.SED()`
- `simcado.source.source_from_image()`
- `simcado.source.Source`

5.12 Keywords for Controlling SimCADO

5.12.1 Observation Parameters

```
Keyword                 Default     [units] Explanation
-------------------------------------------------------------------------------------
↪---------

OBS_DATE                0                       # [dd/mm/yyyy] Date of the␣
↪observation [not yet implemented]
OBS_TIME                0                       # [hh:mm:ss] Time of the observation␣
↪[not yet implemented]
OBS_RA                  90.                     # [deg] RA of the object
OBS_DEC                 -30.                    # [deg] Dec of the object
OBS_ALT                 0                       # [deg] Altitude of the object [not␣
↪yet implemented]
OBS_AZ                  0                       # [deg] Azimuth of the object [not␣
↪yet implemented]
OBS_ZENITH_DIST         0                       # [deg] from zenith
OBS_PARALLACTIC_ANGLE   0                       # [deg] rotation of the source␣
↪relative to the zenith
OBS_SEEING              0.6                     # [arcsec]

OBS_FIELD_ROTATION      0                       # [deg] field rotation with respect␣
↪to the detector array

OBS_DIT                 60                      # [sec] simulated exposure time
OBS_NDIT                1                       # [#] number of exposures taken
OBS_NONDESTRUCT_TRO     2.6                     # [sec] time between non-destructive␣
↪readouts in the detector
OBS_REMOVE_CONST_BG     no                      # remove the median background value
OBS_READ_MODE           single                  # [single, fowler, ramp] Only single␣
↪is implemented at the moment
OBS_SAVE_ALL_FRAMES     no                      # yes/no to saving all DITs in an␣
↪NDIT sequence

OBS_INPUT_SOURCE_PATH   none                    # Path to input Source FITS file
OBS_FITS_EXT            0                       # the extension number where the␣
↪useful data cube is

OBS_OUTPUT_DIR          "./output.fits"         # [filename] Path to save output in.
```

5.12.2 Simulation Parameters

```
Keyword                 Default     [units] Explanation
-------------------------------------------------------------------------------------
↪---------

SIM_DETECTOR_PIX_SCALE  0.004                   # [arcsec] plate scale of the detector
SIM_OVERSAMPLING        1                       # The factor of oversampling inside␣
↪the simulation
SIM_PIXEL_THRESHOLD     1                       # photons per pixel summed over the␣
↪wavelength range. Values less than this are assumed to be zero

SIM_LAM_TC_BIN_WIDTH    0.001                   # [um] wavelength resolution of␣
↪spectral curves
SIM_SPEC_MIN_STEP       1E-4                    # [um] minimum step size where␣
↪resampling spectral curves

SIM_FILTER_THRESHOLD    1E-9                    # transmission below this threshold␣
↪is assumed to be 0
SIM_USE_FILTER_LAM      yes                     # [yes/no] to basing the wavelength␣
↪range off the filter non-zero range - if no, specify LAM_MIN, LAM_MAX
# if "no"
SIM_LAM_MIN             1.9                     # [um] lower wavelength range of␣
↪observation
SIM_LAM_MAX             2.41                    # [um] upper wavelength range of␣
↪observation
SIM_LAM_PSF_BIN_WIDTH   0.1                     # [um] wavelength resolution of the␣
↪PSF layers
SIM_ADC_SHIFT_THRESHOLD 1                       # [pixel] the spatial shift before a␣
↪new spectral layer is added (i.e. how often the spectral domain is sampled for an␣
↪under-performing ADC)

SIM_PSF_SIZE            1024                    # size of PSF
SIM_PSF_OVERSAMPLE      no                      # use astropy's inbuilt oversampling␣
↪technique when generating the PSFs. Kills memory for PSFs over 511 x 511
SIM_VERBOSE             no                      # [yes/no] print information on the␣
↪simulation run
SIM_SIM_MESSAGE_LEVEL   3                       # the amount of information printed␣
↪[5-everything, 0-nothing]

SIM_OPT_TRAIN_IN_PATH   none                    # Options for saving and reusing␣
↪optical trains. If "none": "./"
SIM_OPT_TRAIN_OUT_PATH  none                    # Options for saving and reusing␣
↪optical trains. If "none": "./"
SIM_DETECTOR_IN_PATH    none                    # Options for saving and reusing␣
↪detector objects. If "none": "./"
SIM_DETECTOR_OUT_PATH   none                    # Options for saving and reusing␣
↪detector objects. If "none": "./"
```

5.12.3 Atmospheric Parameters

```
Keyword                 Default     [units] Explanation
-------------------------------------------------------------------------------------
↪---------
```

(continues on next page)

(continued from previous page)

```
ATMO_USE_ATMO_BG        yes                     # [yes/no]

ATMO_TC                 TC_sky_25.tbl           # [filename] for atmospheric␣
↪transmission curve. Default: <pkg_dir>/data/TC_sky_25.tbl
ATMO_EC                 EC_sky_25.tbl           # [filename, "none"] for atmospheric␣
↪emission curve. Default: <pkg_dir>/data/EC_sky_25.tbl
# If ATMO_EC is "none": set ATMO_BG_MAGNITUDE for the simulation filter.
ATMO_BG_MAGNITUDE       13.6                    # [ph/s] background photons for the␣
↪bandpass. If set to None, the ATMO_EC spectrum is assumed to return the needed␣
↪number of photons

ATMO_TEMPERATURE        0                       # deg Celcius
ATMO_PRESSURE           750                     # millibar
ATMO_REL_HUMIDITY       60                      # %
ATMO_PWV                2.5                     # [mm] Paranal standard value
```

5.12.4 Telescope Parameters

```
Keyword                 Default     [units] Explanation
-------------------------------------------------------------------------------------
↪---------

SCOPE_ALTITUDE          3060                    # meters above sea level
SCOPE_LATITUDE          -24.589167              # decimal degrees
SCOPE_LONGITUDE         -70.192222              # decimal degrees

SCOPE_PSF_FILE          scao                    # [scao (default), <filename>, ltao,␣
↪mcao, poppy] import a PSF from a file.
SCOPE_STREHL_RATIO      1                       # [0..1] defines the strength of the␣
↪seeing halo if SCOPE_PSF_FILE is "default"
SCOPE_AO_EFFECTIVENESS  100                     # [%] percentage of seeing PSF␣
↪corrected by AO - 100% = diff limited, 0% = 0.8" seeing
SCOPE_JITTER_FWHM       0.001                   # [arcsec] gaussian telescope jitter␣
↪(wind, tracking)
SCOPE_DRIFT_DISTANCE    0                       # [arcsec/sec] the drift in tracking␣
↪by the telescope
SCOPE_DRIFT_PROFILE     linear                  # [linear, gaussian] the drift␣
↪profile. If linear, simulates when tracking is off. If gaussian, simulates rms␣
↪distance of tracking errors

SCOPE_USE_MIRROR_BG     yes                     # [yes/no]

SCOPE_NUM_MIRRORS       5                       # number of reflecting surfaces
SCOPE_TEMP              0                       # deg Celsius - temperature of mirror
SCOPE_M1_TC             TC_mirror_EELT.dat      # [filename] Mirror reflectance curve.
↪ Default: <pkg_dir>/data/TC_mirror_EELT.dat
SCOPE_MIRROR_LIST       EC_mirrors_EELT_SCAO.tbl    # [filename] List of mirror sizes.
↪     Default: <pkg_dir>/data/EC_mirrors_EELT_SCAO.tbl
```

5.12.5 Instrument Parameters

```
Keyword                 Default     [units] Explanation
-------------------------------------------------------------------------------------
↪---------
```

(continues on next page)

(continued from previous page)

```
INST_TEMPERATURE        -190                    # deg Celsius - inside temp of␣
↪instrument

INST_ENTR_NUM_SURFACES  4                       # number of surfaces on the entrance␣
↪window
INST_ENTR_WINDOW_TC     TC_window.dat           # [filename] Default: <pkg_dir>/data/
↪TC_window.dat --> transmission = 0.98 per surface

INST_DICHROIC_NUM_SURFACES  2                   # number of surfaces on the entrance␣
↪window
INST_DICHROIC_TC        TC_dichroic.dat         # [filename] Default: <pkg_dir>/data/
↪TC_dichroic.dat --> transmission = 1 per surface

INST_FILTER_TC          Ks                      # [filename, string(filter name)]␣
↪List acceptable filters with >>> simcado.optics.get_filter_set()

INST_PUPIL_NUM_SURFACES 2                       # number of surfaces on the pupil␣
↪window
INST_PUPIL_TC           TC_pupil.dat            # [filename] Default: <pkg_dir>/data/
↪TC_pupil.dat --> transmission = 1 per surface

# MICADO, collimator 5x, wide-field 2x (zoom 4x), camera 4x
INST_NUM_MIRRORS        11                      # number of reflecting surfaces in␣
↪MICADO
INST_MIRROR_TC          TC_mirror_gold.dat      # [filename, "default"] If "default":␣
↪INST_MIRROR_TC = SCOPE_M1_TC

INST_USE_AO_MIRROR_BG   yes                     # [yes/no]
INST_AO_TEMPERATURE     0                       # deg Celsius - inside temp of AO␣
↪module
INST_NUM_AO_MIRRORS     7                       # number of reflecting surfaces␣
↪between telescope and instrument (i.e. MAORY)
INST_MIRROR_AO_TC       TC_mirror_gold.dat      # [filename, "default"] If "default":␣
↪INST_MIRROR_AO_TC = INST_MIRROR_TC
INST_MIRROR_AO_LIST     EC_mirrors_ao.tbl       # List of mirrors in the AO. Default:
↪<pkg_dir>/data/EC_mirrors_ao.tbl

INST_ADC_PERFORMANCE    100                     # [%] how well the ADC does its job
INST_ADC_NUM_SURFACES   8                       # number of surfaces in the ADC
INST_ADC_TC             TC_ADC.dat              # [filename] Default: <pkg_dir>/data/
↪TC_ADC.dat --> transmission = 0.98 per surface

INST_DEROT_PERFORMANCE  100                     # [%] how well the derotator derotates
INST_DEROT_PROFILE      linear                  # [linear, gaussian] the profile with␣
↪which it does it's job

INST_DISTORTION_MAP     none                    # path to distortion map
INST_WFE                data/INST_wfe.tbl       # [nm] (float or filename) A single␣
↪number for the total WFE of a table of wavefront errors for each surface in the␣
↪instrument
INST_FLAT_FIELD         none                    # path to a FITS file containing a␣
↪flat field (median = 1) for each chip.
```

5.12.6 Spectroscopy parameters

```
Keyword                  Default      [units] Explanation
-------------------------------------------------------------------------------------
↪---------

SPEC_ORDER_SORT          HK                      # Order-sorting filter: "HK" or "IJ"
SPEC_SLIT_WIDTH          narrow                  # Slit width: "narrow" or "wide"
```

5.12.7 Detector parameters

```
Keyword                  Default      [units] Explanation
-------------------------------------------------------------------------------------
↪---------

FPA_USE_NOISE            yes                     # [yes/no]

FPA_READOUT_MEDIAN       4                       # e-/px
FPA_READOUT_STDEV        1                       # e-/px
FPA_DARK_MEDIAN          0.01                    # e-/s/px
FPA_DARK_STDEV           0.01                    # e-/s/px

FPA_QE                   TC_detector_H2RG.dat    # [filename] Quantum efficiency of␣
↪detector.
FPA_NOISE_PATH           FPA_noise.fits          # [filename, "generate"] if "generate
↪": use NGHxRG to create a noise frame.
FPA_GAIN                 1                       # e- to ADU conversion
FPA_LINEARITY_CURVE      FPA_linearity.dat       # [filename, "none"]
FPA_FULL_WELL_DEPTH      1E5                     # [e-] The level where saturation␣
↪occurs

FPA_PIXEL_MAP            none                    # path to a FITS file with the pixel␣
↪sensitivity map
# if FPA_PIXEL_MAP == none
FPA_DEAD_PIXELS          1                       # [%] if FPA_PIXEL_MAP=none, a␣
↪percentage of detector pixel which are dead
FPA_DEAD_LINES           1                       # [%] if FPA_PIXEL_MAP=none, a␣
↪percentage of detector lines which are dead

FPA_CHIP_LAYOUT          centre                    # ["tiny", "small", "centre", "full
↪", <filename>] description of the chip layout on the detector array.
FPA_PIXEL_READ_TIME      1E-5                    # [s] read time for y pixel -␣
↪typically ~10 us
FPA_READ_OUT_SCHEME      double_corr             # "double_corr", "up-the-ramp",
↪"fowler"
```

5.12.8 NXRG Noise Generator package parameters

```
Keyword                  Default      [units] Explanation
-------------------------------------------------------------------------------------
↪---------
# See Rauscher (2015) for details
# http://arxiv.org/pdf/1509.06264.pdf
```

(continues on next page)

(continued from previous page)

```
HXRG_NUM_OUTPUTS        64                      # Number of
HXRG_NUM_ROW_OH         8                       # Number of row overheads
HXRG_PCA0_FILENAME      FPA_nirspec_pca0.fits   # if "default": <pkg_dir>/data/
HXRG_OUTPUT_PATH        none                    # Path to save the detector noise
HXRG_PEDESTAL           4                       # Pedestal noise
HXRG_CORR_PINK          3                       # Correlated Pink noise
HXRG_UNCORR_PINK        1                       # Uncorrelated Pink noise
HXRG_ALT_COL_NOISE      0.5                     # Alternating Column noise

HXRG_NAXIS1             4096                    # Size of the HAWAII 4RG detectors
HXRG_NAXIS2             4096
HXRG_NUM_NDRO           1                       # Number of non-destructive readouts␣
↪to add to a noise cube
```

5.13 SimCADO Package

5.13.1 simcado.commands module

simcado.commands Module

Functions

`dump_defaults`([filename, selection])	Dump the frequent.config file to a path specified by the user
`dump_chip_layout`([path])	Dump the FPA_chip_layout.dat file to a path specified by the user
`dump_mirror_config`([path, what])	Dump the EC_mirrors_scope.tbl or the EC_mirrors_ao.tbl to disk
`read_config`(config_file)	Read in a SimCADO configuration file
`update_config`(config_file, config_dict)	Update a SimCADO configuration dictionary

dump_defaults

`simcado.commands.`**`dump_defaults`**(*filename=None*, *selection='freq'*)

Dump the frequent.config file to a path specified by the user

Parameters

filename [str, optional] path or filename where the .config file is to be saved

selection [str, optional] ["freq", "all"] amount of keywords to save. "freq" only prints the most frequently used keywords. "all" prints all of them

dump_chip_layout

`simcado.commands.`**`dump_chip_layout`**(*path=None*)

Dump the FPA_chip_layout.dat file to a path specified by the user

Parameters

path [str, optional] path where the chip layout file is to be saved

dump_mirror_config

simcado.commands.**dump_mirror_config**(*path=None*, *what='scope'*)

Dump the EC_mirrors_scope.tbl or the EC_mirrors_ao.tbl to disk

Parameters

path [str, optional] path where the mirror configuration file is to be saved

what [str, optional] ["scope", "ao"] dump the mirror configuration for either the telescope or the AO module

read_config

simcado.commands.**read_config**(*config_file*)

Read in a SimCADO configuration file

The configuration file is in SExtractor format: 'PARAMETER Value # Comment'

Parameters

config_file [str] the filename of the .config file

Returns

config_dict [dict (collections.OrderedDict)] A dictionary with keys 'PARAMETER' and values 'Value'.

Notes

The values of the dictionary are strings and will have to be converted to the appropriate data type as they are needed.

update_config

simcado.commands.**update_config**(*config_file*, *config_dict*)

Update a SimCADO configuration dictionary

A configuration file in the SExtractor format:

```
'PARAMETER    Value    # Comment'
```

an existing configuration dictionary.

Parameters

config_file [str] the filename of the .config file

Returns

config_dict [dict] A dictionary with keys 'PARAMETER' and values 'Value'.

Notes

the values of the dictionary are strings and will have to be converted to the appropriate data type as they are needed.

Classes

`UserCommands`([filename, sim_data_dir])	An extended dictionary with the parameters needed for running a simulation

UserCommands

class `simcado.commands.`**`UserCommands`**(*filename=None*, *sim_data_dir=None*)

Bases: `object`

An extended dictionary with the parameters needed for running a simulation

A `UserCommands` object contains a dictionary which holds all the keywords from the `default.config` file. It also has attributes which represent the frequently used variables, i.e. `pix_res`, `lam_bin_edges`, `exptime`, etc

`<UserCommands>.cmds` is a dictionary that holds all the variables the user may wish to change. It also has some set variables like `<UserCommands>.pix_res` that can be accessed directly, instead of from the dictionary.

`UserCommands` is imported directly into the simcado package and is accessable from the main package - `simcado.UserCommands`

If UserCommands is called without any arguments, the default values for MICADO and the E-ELT are used.

Parameters

filename [str, optional] path to the user's .config file

See also:

`simcado.detector.Detector`

`simcado.optics.OpticalTrain`

Examples

By default `UserCommands` contains the parameters needed to generate the MICADO optical train:

```
>>> import simcado
>>> my_cmds = simcado.UserCommands()
```

To list the keywords that are available:

```
>>> my_cmds.keys()
...
```

The `UserCommands` object also contains smaller dictionaries for each category of keywords - e.g. for the keywords describing the instrument:

```
>>> my_cmds.inst
...
```

Attributes

Internal dictionaries

cmds [dict (collections.OrderedDict)] the dictionary which holds all the keyword-value pairs needed for running a simualtion

obs [dict (collections.OrderedDict)] parameters about the observation

sim [dict (collections.OrderedDict)] parameters about the simualtion

atmo [dict (collections.OrderedDict)] parameters about the atmosphere

scope [dict (collections.OrderedDict)] parameters about the telescope

inst [dict (collections.OrderedDict)] parameters about the instrument

fpa [dict (collections.OrderedDict)] parameters about the detector array (FPA - Focal Plane Array)

hxrg [dict (collections.OrderedDict)] parameters about the chip noise (HxRG - HAWAII 4RG chip series)

Attributes pertaining to the purely spectral data sets (e.g. transmission

curves, stellar spectra, etc)

lam [np.ndarray] a vector containing the centres of the wavelength bins used when resampling the spectra or transmission curves

lam_res [float] [um] the resolution of the `lam`

Attributes pertaining to the binning in spectral space for which different

PSFs need to be used

lam_psf_res [float] [um] the spectal "distance" between layers - i.e. width of the bins

lam_bin_edges [array-like] [um] the edge of the spectral bin for each layer

lam_bin_centers [array-like] [um] the centres of the spectral bin for each layer

Attributes pertaining to the binning in the spatial plane

pix_res [float] [arcsec] the internal (oversampled) spatial resolution of the simulation

fpa_res [float] [arcsec] the field of view of the individual pixels on the detector

General attributes

verbose [bool] Flag for printing intermediate results to the screen (default=True)

exptime [float] [s] exposure time of a single DIT

diameter [float] [m] outer diamter of the primary aperture (i.e. M1)

area [float] [m^2] effective area of the primary aperture (i.e. M1)

filter [str] [BVRIzYJHKKs,user] filter used for the observation

Methods

update(new_dict)	updates the current `UserCommands` object from another dict-like object
keys()	returns the keys in the `UserCommands.cmds` dictionary
values()	returns the values in the `UserCommands.cmds` dictionary

Methods Summary

`keys`(self)	Return the keys in the *UserCommands.cmds* dictionary
`update`(self, new_dict)	Update multiple entries of a `UserCommands` dictionary
`values`(self)	Return the values in the *UserCommands.cmds* dictionary
`writeto`(self[, filename])	Write all the key-value commands to an ASCII file on disk

Methods Documentation

keys(*self*)

Return the keys in the *UserCommands.cmds* dictionary

update(*self*, *new_dict*)

Update multiple entries of a `UserCommands` dictionary

`update(new_dict)` takes either a normal python `dict` object or a `UserCommands` object. Only keywords that match those in the `UserCommands` object will be updated. Misspelled keywords raise an error.

To update single items in the dictionary, it is recommended to simply call the key and update the value - i.e `<UserCommands>[key] = value`.

Parameters

new_dict [dict, `UserCommands`]

Raises

KeyError If a parameter is not found in `self.cmds`.

See also:

`UserCommands`

Examples

View the default commands

```
>>> import simcado
>>> my_cmds = simcado.UserCommands()
>>> print(my_cmds.cmds)
```

Change a single command

```
>>> my_cmds["OBS_DIT"] = 60
```

Change a series of commands at once

```
>>> new_cmds = {"OBS_DIT" : 60 , "OBS_NDIT" : 10}
>>> my_cmds.update(new_cmds)
```

values (*self*)
Return the values in the *UserCommands.cmds* dictionary

writeto (*self, filename='commands.config'*)
Write all the key-value commands to an ASCII file on disk

Parameters

filename [str] file path for where the file should be saved

5.13.2 `simcado.detector` module

simcado.detector Module

A description of the Chip noise properties and their positions on the Detector

Module Summary

This module holds three classes: `Detector`, `Chip` and `HXRGNoise`.

`Chip` Everything to do with photons and electrons happens in the `Chip` class. Each `Chip` is initialised with a position relative to the centre of the detector array, a size [in pixels] and a resolution [in arcsec]. Photons fall onto the `Chip` s and are read out together with the read noise characteristics of the `Chip`.

`Detector` The `Detector` holds the information on where the `Chip` s are placed on the focal plane. Focal plane coordinates are in [arcsec]. These coordinates are either read in from a default file or determined by the user. The `Detector` object is an intermediary - it only passes information on the photons to the `Chip` s. It is mainly a convenience class so that the user can read out all `Chip` s at the same time.

Classes

Detector builds an array of `Chip` objects based on a `UserCommands` object

Chip converts incoming photons into ADUs and adds in read-out noise

See Also

`simcado.optics.OpticalTrain`, `simcado.source.Source`

Notes

References

[1] Bernhard Rauscher's HxRG Noise Generator script

Examples

The Detector can be used in stand-alone mode. In this case it outputs only the noise that a sealed-off detector would generate:

```
>>> import simcado
>>> fpa = simcado.Detector(simcado.UserCommands())
>>> fpa.read_out(output=True, chips=[0])
```

The Detector is more useful if we combine it with a Source object and an OpticalTrain. Here we create a Source object for an open cluster in the LMC and pass the photons arriving from it through the E-ELT and MICADO. The photons are then cast onto the detector array. Each Chip converts the photons to ADUs and adds the resulting image to an Astropy HDUList. The HDUList is then written to disk.

```
>>> # Create a set of commands, optical train and detector
>>>
>>> import simcado
>>> cmds = simcado.UserCommands()
>>> opt_train = simcado.OpticalTrain(cmds)
>>> fpa = simcado.Detector(cmds)
>>>
>>> # Pass photons from a 10^4 Msun open cluster in the LMC through to the detector
>>>
>>> src = sim.source.source_1E4_Msun_cluster()
>>> src.apply_optical_train(opt_train, fpa)
>>>
>>># Read out the detector array to a FITS file
>>>
>>> fpa.read_out(filename="my_raw_image.fits")
```

Functions

open(self, filename)	Opens a saved Detector file.
plot_detector(detector)	Plot the contents of a detector array
plot_detector_layout(detector[, plane, fmt, …])	Plot the detector layout
make_noise_cube([num_layers, filename, …])	Create a large noise cube with many separate readout frames.
install_noise_cube([n])	Install a noise cube in the package directory

open

simcado.detector.**open**(*self*, *filename*)

Opens a saved Detector file.

** Not yet implemented ** ** Should be moved outside of Detector and called with detector.open() **

Detector objects can be saved to FITS file and read back in for later simulations.

Parameters

filename [str] path to the FITS file where the Detector object is stored

Returns

`simcado.Detector` **object**

plot_detector

`simcado.detector.`**`plot_detector`**`(`*detector*`)`

Plot the contents of a detector array

Parameters

detector [simcado.Detector] The detector object to be shown

plot_detector_layout

`simcado.detector.`**`plot_detector_layout`**`(`*detector*, *plane='sky'*, *fmt='g-'*, *plot_origin=False*, *label=True*, ***kwargs*`)`

Plot the detector layout

Parameters

detector [simcado.Detector] The Detector to be shown

plane ['sky' or 'fpa'] Plot detector layout on the sky (in arcsec) or in the focal plane (in mm)

fmt [matplotlib format string]

plot_origin [bool] Mark position pixel (1,1) for each chip.

label [bool] Label the chips with their numberq

make_noise_cube

`simcado.detector.`**`make_noise_cube`**`(`*num_layers=25*, *filename='FPA_noise.fits'*, *multicore=True*`)`

Create a large noise cube with many separate readout frames.

Note: Each frame takes about 15 seconds to be generated. The default value of 25 frames will take around six minutes depending on your computer's architecture.

Parameters

num_layers [int, optional] the number of separate readout frames to be generated. Default is 25.

filename [str, optional] The filename for the FITS cube. Default is "FPA_noise.fits"

multicore [bool, optional] If you're not using windows, this allows the process to use all available cores on your machine to speed up the process. Default is True

Notes

multicore doesn't work - fix it

install_noise_cube

`simcado.detector.`**`install_noise_cube`**`(`*n=9*`)`

Install a noise cube in the package directory

Parameters

n [int, optional] number of layers.

Classes

`Detector`(cmds[, small_fov])	Generate a series of `Chip` objects for a focal plane array
`Chip`(x_cen, y_cen, x_len, y_len, pix_res[, ...])	Holds the "image" as seen by a single chip in the focal plane

Detector

class `simcado.detector.`**`Detector`**(*cmds*, *small_fov=True*)

Bases: `object`

Generate a series of `Chip` objects for a focal plane array

The `Detector` is a holder for the series of `Chip` objects which make up the detector array. The main advantage of the `Detector` object is that the user can read out all chips in the whole detector array at once. A `Detector` is a parameter in the `Source.apply_optical_train()` method.

Parameters

cmds [UserCommands] Commands for how to model the Detector

small_fov [bool, optional] Default is True. Uses a single 1024x1024 window at the centre of the FoV

See also:

`Chip`, **`Source`**

`OpticalTrain`, **`UserCommands`**

Examples

Create a `Detector` object

```
>>> import simcado as sim
>>> my_cmds = sim.UserCommands()
>>> my_detector = sim.Detector(my_cmds)
```

Read out only the first `Chip`

```
>>> my_detector.readout(filename=image.fits, chips=[0])
```

Attributes

cmds [UserCommands] commands for modelling the detector layout and exposures

layout [astropy.table.Table] table of positions and sizes of the chips on the focal plane

chips [list] a list of the `Chips` which make up the detector array

oversample [int] factor between the internal angular resolution and the pixel FOV

fpa_res [float] [mas] field of view of a single pixel

dit [float] [s] exposure time of a single DIT

tro [float] [s] time between consecutive non-destructive readouts in up-the-ramp mode

ndit [int] number of exposures (DITs)

Methods

read_out()	for reading out the detector array into a FITS file
open()	not yet implemented
write()	not yet implemented Save the Detector object into a FITS file
.. todo::	Open should be moved into a general function for detector.py which returns a `Detector` object after reading in a saved detector file

Methods Summary

`read_out`(self[, filename, to_disk, chips, . . .])	Simulate the read-out process of the detector array
`write`(self[, filename])	Write a `Detector` object out to a FITS file

Methods Documentation

`read_out` (*self*, *filename=None*, *to_disk=False*, *chips=None*, *read_out_type='superfast'*, ***kwargs*)
Simulate the read-out process of the detector array

Based on the parameters set in the `UserCommands` object, the detector will read out the images stored on the `Chips` according to the specified read-out scheme, i.e. Fowler, up-the-ramp, single read, etc.

Parameters

filename [str] where the file is to be saved. If `None` and `to_disk` is true, the output file is called "output.fits". Default is `None`

to_disk [bool] a flag for where the output should go. If `filename` is given or if `to_disk=True`, the `Chip` images will be written to a *.fits*' file on disk. If no *filename*' is specified, the output will be called "output.fits".

chips [int, array-like, optional] The chip or chips to be read out, based on the detector_layout.dat file. Default is the first `Chip` specified in the list, i.e. [0].

read_out_type [str, optional] The name of the algorithm used to read out the chips: - "superfast" - "non_destructive" - "up_the_ramp"

Returns

astropy.io.fits.HDUList

`write` (*self*, *filename=None*, ***kwargs*)
Write a `Detector` object out to a FITS file

Writes the important information contained in a `Detector` object into FITS file for later use. The main information written out includes: the layout of the detector chips, any pixel maps associated with the detector chips, a linearity curve and a QE curve for the chips.

Parameters

filename [str, optional] path to the FITS file where the `Detector` object is stored. If `filename=None` (by default), the file written is `./detector.fits`

Chip

class simcado.detector.**Chip**(*x_cen, y_cen, x_len, y_len, pix_res, pixsize=15, angle=0, gain=1, obs_coords=[0, 0], fieldangle=0, chipid=None, flat_field=None*)

Bases: `object`

Holds the "image" as seen by a single chip in the focal plane

The `Chip` object contains information on where it is located in the focal plane array. The method `<Source>.apply_optical_train()` passes an image of the on-sky object to each `Chip`. This image is resampled to the `Chip` pixel scale. Each `Chip` holds the "ideal" image as an array of expectation values for number of photons arriving per second. The `Chip` then adds detector noise and other characteristics to the image when <Detector>.readout() is called.

Parameters

x_cen, y_cen [float] [micron] the coordinates of the centre of the chip relative to the centre of the focal plane

x_len, y_len [int] the number of pixels per dimension

pix_res [float] [arcsec] the field of view per pixel

id [int] an identification number for the chip (assuming they are not correctly ordered)

flat_field [np.ndarray] a 2D array holding the flat fielding effects for the chip

See also:

`Detector`

`simcado.source.Source`

`simcado.commands.UserCommands`

`simcado.optics.OpticalTrain`

Attributes

x_cen, y_cen [float] [arcsec] the coordinates of the centre of the chip relative to the centre of the focal plane

naxis1, naxis2 [int] the number of pixels per dimension

pix_res [float] [arcsec] the field of view per pixel

chipid [int, optional] the id of the chip relative to the others on the detector array. Default is `None`

dx, dy [float] [arcsec] half of the field of view of each chip

x_min, x_max, y_min, y_max [float] [arcsec] the borders of the chip relative to the centre of the focal plane

array [np.ndarray] an array for holding the signal registered by the `Chip`

Methods

add_signal(signal)	adds signal to `.array`. The signal should be the same dimensions as `Chip.array`
add_uniform_background(emission, lam_min, lam_max, output=False)	adds a constant to the signal in `.array`. The background level is found by integrating the `emission` curve between `lam_min` and `lam_max`. If output is set to `True`, an image with the same dimensions as `.array` scaled to the background flux is returned.
apply_pixel_map(pixel_map_path=None, dead_pix=None, max_well_depth=1E5)	applies a mask to `.array` representing the position of the current "hot" and "dead" pixels / lines
reset()	resets the signal on the `Chip` to zero. In future releases, an implementation of the persistence characteristics of the detector will go here.

Methods Summary

`add_signal`(self, signal)	Add a 2D array of photon signal to the Chip
`add_uniform_background`(self, emission, . . .)	Add a uniform background
`apply_pixel_map`(self[, pixel_map_path, . . .])	Adds "hot" and "dead" pixels to the array
`read_out`(self, cmds[, read_out_type])	Read out the detector array
`reset`(self)	

Methods Documentation

`add_signal`(*self*, *signal*)

Add a 2D array of photon signal to the Chip

Add some signal photons to the detector array. Input units are expected to be [ph/s/pixel]

Parameters

signal [np.ndarray] [ph/pixel/s] photon signal. `signal` should have the same dimensions as the `array`

Returns

None

`add_uniform_background`(*self*, *emission*, *lam_min*, *lam_max*, *output=False*)

Add a uniform background

Take an EmissionCurve and some wavelength boundaries, lam_min and lam_max, and sum up the photons in between. Add those to the source array.

Parameters

- emission_curve: EmissionCurve object with background emission photons

- lam_min, lam_max: the wavelength limits

Optional keywords:

- output: [False, True] if output is True, the BG emission array is returned

Output is in [ph/s/pixel].

apply_pixel_map(*self*, *pixel_map_path=None*, *dead_pix=None*, *max_well_depth=100000.0*)
Adds "hot" and "dead" pixels to the array

applies a mask to `.array` representing the positions of the current "hot" and "dead" pixels / lines. The method either reads in a FITS file with locations of these pixels, or generates a series of random coordinates and random weights for the pixels.

Parameters

pixel_map_path [str] path to the FITS file. Default is None

dead_pix [int] [%] the percentage of dead or hot pixels on the chip - only used if `pixel_map_path = None`. Default is `None`.

max_well_depth [1E5]

Returns

None

read_out(*self*, *cmds*, *read_out_type='superfast'*)
Read out the detector array

Parameters

cmds [simcado.UserCommands] Commands for how to read out the chip

Returns

out_array [np.ndarray] image of the chip read out

reset(*self*)

5.13.3 simcado.nghxrg module

simcado.nghxrg Module

NGHXRG by Bernard Rauscher see the paper: http://arxiv.org/abs/1509.06264 downloaded from: http://jwst.nasa.gov/publications.html

Classes

`HXRGNoise`([naxis1, naxis2, naxis3, n_out, . . .])	A class to generate HxRG noise frames

HXRGNoise

class simcado.nghxrg.**HXRGNoise**(*naxis1=None*, *naxis2=None*, *naxis3=None*, *n_out=None*, *dt=None*, *nroh=None*, *nfoh=None*, *pca0_file=None*, *verbose=False*, *reverse_scan_direction=False*, *reference_pixel_border_width=None*)
Bases: `object`

A class to generate HxRG noise frames

HXRGNoise is a class for making realistic Teledyne HxRG system noise. The noise model includes correlated, uncorrelated, stationary, and non-stationary components. The default parameters make noise that resembles Channel 1 of JWST NIRSpec. NIRSpec uses H2RG detectors. They are read out using four video outputs at 100.000 pix/s/output.

Attributes Summary

`nghxrg_version`

Methods Summary

`message`(self, message_text)	Used for status reporting
`mknoise`(self, o_file[, rd_noise, pedestal, …])	Generate a FITS cube containing only noise.
`pink_noise`(self, mode)	Generate a vector of non-periodic pink noise.
`white_noise`(self[, nstep])	Generate white noise for an HxRG including all time steps (actual pixels and overheads).

Attributes Documentation

`nghxrg_version = 2.3`

Methods Documentation

`message`(*self*, *message_text*)

Used for status reporting

`mknoise`(*self*, *o_file*, *rd_noise=None*, *pedestal=None*, *c_pink=None*, *u_pink=None*, *acn=None*, *pca0_amp=None*, *reference_pixel_noise_ratio=None*, *ktc_noise=None*, *bias_offset=None*, *bias_amp=None*)

Generate a FITS cube containing only noise.

Parameters

o_file [str] Output filename

pedestal [float] Magnitude of pedestal drift in electrons

rd_noise [float] Standard deviation of read noise in electrons

c_pink [float] Standard deviation of correlated pink noise in electrons

u_pink [float] Standard deviation of uncorrelated pink noise in electrons

acn: float Standard deviation of alterating column noise in electrons

pca0 [float] Standard deviation of pca0 in electrons

reference_pixel_noise_ratio [float] Ratio of the standard deviation of the reference pixels to the regular pixels. Reference pixels are usually a little lower noise.

ktc_noise [float] kTC noise in electrons. Set this equal to sqrt(k*T*C_pixel)/q_e, where k is Boltzmann's constant, T is detector temperature, and C_pixel is pixel capacitance. For an H2RG, the pixel capacitance is typically about 40 fF.

bias_offset [float] On average, integrations stare here in electrons. Set this so that all pixels are in range.

bias_amp [float] A multiplicative factor that we multiply PCA-zero by to simulate a bias pattern. This is completely independent from adding in "picture frame" noise.

Notes

Because of the noise correlations, there is no simple way to predict the noise of the simulated images. However, to a crude first approximation, these components add in quadrature.

The units in the above are mostly "electrons". This follows convention in the astronomical community. From a physics perspective, holes are actually the physical entity that is collected in Teledyne's p-on-n (p-type implants in n-type bulk) HgCdTe architecture.

pink_noise(*self*, *mode*)
Generate a vector of non-periodic pink noise.

Parameters

mode [str] Selected from {'pink', 'acn'}

white_noise(*self*, *nstep=None*)
Generate white noise for an HxRG including all time steps (actual pixels and overheads).

Parameters

nstep [int] Length of vector returned

5.13.4 simcado.optics module

simcado.optics Module

optics.py

Functions

get_filter_curve(filter_name)	Return a Vis/NIR broadband filter TransmissionCurve object
get_filter_set([path])	Return a list of the filters installed in the package directory

get_filter_curve

simcado.optics.**get_filter_curve**(*filter_name*)
Return a Vis/NIR broadband filter TransmissionCurve object

Parameters

filter_name [str]

Notes

Acceptable filters can be found be calling get_filter_set()

To access the values use TransmissionCurve.lam and TransmissionCurve.val

Examples

```
>>> transmission_curve = get_filter_curve("TC_filter_Ks.dat")
>>> wavelength   = transmission_curve.lam
>>> transmission = transmission_curve.val
```

get_filter_set

`simcado.optics.`**`get_filter_set`**(*path=None*)

Return a list of the filters installed in the package directory

Classes

`OpticalTrain`(cmds, **kwargs)	The OpticalTrain object reads in or generates the information necessary to model the optical path for all (3) sources of photons: the astronomical source, the atmosphere and the primary mirror.

OpticalTrain

class `simcado.optics.`**`OpticalTrain`**(*cmds*, ***kwargs*)

Bases: `object`

The OpticalTrain object reads in or generates the information necessary to model the optical path for all (3) sources of photons: the astronomical source, the atmosphere and the primary mirror.

Parameters

cmds [UserCommands, optional] Holds the commands needed to generate a model of the optical train

See also:

`simcado.commands.UserCommands`

`simcado.commands.dump_defaults()`

Methods Summary

`apply_derotator`(self, arr)	
`apply_tracking`(self, arr)	
`apply_wind_jitter`(self, arr)	
`read`(self, filename)	
`replace_psf`(self, new_psf, lam_bin_centers)	Change the PSF of the optical train
`save`(self, filename)	
`update_filter`(self[, trans, lam, filter_name])	Update the filter curve without recreating the full OpticalTrain object

Methods Documentation

apply_derotator (*self*, *arr*)

apply_tracking (*self*, *arr*)

apply_wind_jitter (*self*, *arr*)

read (*self*, *filename*)

replace_psf (*self*, *new_psf*, *lam_bin_centers*)
Change the PSF of the optical train

save (*self*, *filename*)

update_filter (*self*, *trans=None*, *lam=None*, *filter_name=None*)
Update the filter curve without recreating the full OpticalTrain object

Parameters

trans [TransmissionCurve, np.array, list, optional] [0 .. 1] the transmission coefficients. Either a TransmissionCurve object can be passed (in which case omit `lam`) or an array/list can be passed (in which case specify `lam`)

lam [np.array, list, optional] [um] an array for the spectral bin centres, if `trans` is not a TransmissionCurve object

filter_name [str, optional] The name of a filter curve contained in the package_dir. User get_filter_set() to find which filter curves are installed.

See also:

simcado.spectral.TransmissionCurve

simcado.optics.get_filter_set()

5.13.5 simcado.psf module

simcado.psf Module

PSFs and PSFCubes

Todo: revise this opening text

Description

Car Sagan said

"If you want to bake an apple pie from scratch, first you must create the universe"

Single PSFs

We need to start by generating a single PSF in order to generate a PSFCube. We need to know the spatial characteristics of the PSF: The commonalities of all PSFs are:

- pix_width
- pix_height
- pix_res
- type

The types of PSF offered: Moffat, Gaussian2D, Airy, Delta, Line, User For each of the PSF types we need to create a subclass of PSF. Each subclass takes its own list of parameters:

- MoffatPSF (alpha, beta)
- GaussianPSF (fwhm, eccentricity=0, angle=0)
- AiryPSF (first_zero, eccentricity=0, angle=0)
- DeltaPSF (x=0, y=0)
- LinePSF (x0, x1, y0, y1, angle=0)
- UserPSFCube (filename, ext_no=0)

Multiple PSFs in a Cube

To generate a PSF cube we need to know the spectral bins and the type of PSF. The bins are defined by a central wavelength, however a cube should also contain the edges of each bin so that transmission and emission can be re-binned properly. - lam_bin_centers - lam_bin_edges - lam_res

A PSF instance will have these additional arguments: - array … a 2D array to hold the PSF image

A psf instance will have these additional arguments: - cube … a (l,x,y) 3D array to hold the PSF cube

As far as input goes, psf should be able to accept a dictionary with the keywords necessary to build the cube.

Notes

All wavelength values are given in [um] All pixel dimensions are given in [arcsec] All angles are given in [deg]

Classes

PSF(object) psf(object)

Subclasses

MoffatPSF(PSF) GaussianPSF(PSF) AiryPSF(PSF) DeltaPSF(PSF) LinePSF(PSF) UserPSFCube(PSF)

Deltapsf(psf) Airypsf(psf) Gaussianpsf(psf) Moffatpsf(psf) CombinedPSFCube(psf) UserPSFCube(psf) ADC_psf(psf)

There are two types of psf object here: - a cube - a single psf image

The cube is essentially a list of psf images, homogenized in size Should we have separate classes for these?

Both PSF and psf can be created from a single model or through convolution of a list of PSF components

Functions

`poppy_eelt_psf`([plan, wavelength, mode, . . .])	Generate a PSF for the E-ELT for plan A or B with POPPY
`poppy_ao_psf`(strehl[, mode, plan, size, . . .])	Create a diffraction limited E-ELT PSF with a Seeing halo
`seeing_psf`([fwhm, psf_type, size, pix_res, . . .])	Return a seeing limited PSF
`get_eelt_segments`([plan, missing, . . .])	Generate a list of segments for POPPY for the E-ELT
`make_foreign_PSF_cube`(fnames[, out_name, . . .])	Combine several PSF FITS images into a single PSF FITS file

poppy_eelt_psf

`simcado.psf.`**`poppy_eelt_psf`**(*plan='A'*, *wavelength=2.2*, *mode='wide'*, *size=1024*, *segments=None*, *filename=None*, *use_pupil_mask=True*, ***kwargs*)

Generate a PSF for the E-ELT for plan A or B with POPPY

Parameters

plan [str, optional] ["A", "B"], Default = "A" * Plan A is for a fully populated mirror (798 segments) * Plan B has the inner 5 rings missing (588 segments) and a further 5 random segments missing (583 segments)

wavelength [float, list, array, optional] [um] Default = 2.2um. The wavelength(s) for which a PSF should be made

mode [str, optional] ["wide", "zoom"] Default = "wide". Sets the pixel size for each of the MICADO imaging modes - {"wide" : 4mas, "zoom" : 1.5mas}

size [int, optional] [pixels] Default = 1024

segments [list, optional] Default = None. A list of which segments to use for generating the E-ELT mirror. See `get_eelt_segments()`

filename [str, optional] Default = None. If filename is not None, the resulting FITS object will be saved to disk

use_pupil_mask [str, optional] Default = True.

Returns

``astropy.HDUList`` [an astropy FITS object with the PSF in the data]

extensions

Other Parameters

Values to pass to the POPPY functions

flattoflat [float] [m] Default : 1.256

gap [float] [m] Default : 0.004

secondary_radius [float] [m] Default : 5

n_supports [int] Default : 6

support_width [float] [m] Default : 0.2

support_angle_offset [float] [deg] Default : 0

n_missing [int] Default : None. Number of segments missing

pupil_inner_radius [float] [m] Default : None # Plan A: 5.6m, Plan B: 11.5m

pupil_outer_radius [float] [m] Default : 19

See also:

`get_eelt_segments()`

poppy_ao_psf

`simcado.psf.`**`poppy_ao_psf`**(*strehl*, *mode='wide'*, *plan='A'*, *size=1024*, *filename=None*, ***kwargs*)

Create a diffraction limited E-ELT PSF with a Seeing halo

Uses POPPY to create a diffraction limited PSF for the E-ELT for a certain configuration of mirror segments. The diffraction limited core is added to seeing halo, modelled by either a moffat or gassian profile.

Parameters

strehl [float] [0 .. 1] The components are summed and weighted according to the strehl ratio psf = (1-strehl)*seeing_psf + (strehl)*diff_limited_psf

mode [str, optional] ["wide", "zoom"] Default = "wide". Sets the pixel size for each of the MICADO imaging modes - {"wide" : 4mas, "zoom" : 1.5mas}

plan [str, optional] ["A", "B"], Default = "A" * Plan A is for a fully populated mirror (798 segments) * Plan B has the inner 5 rings missing (588 segments) and a further 5 random segments missing (583 segments)

size [int, optional] [pixels] Default = 1024

filename [str, optional] Default = None. If filename is not None, the resulting FITS object will be saved to disk

Returns

hdu_list [astropy.HDUList] an astropy FITS object containing the PSFs for the given wavelengths

Other Parameters

fwhm [float] [arcsec] Default : 0.8

psf_type [str] Default : "moffat"

wavelength [float] [um] Default : 2.2

segments [list] Default : None. A list of which hexagonal poppy segments to use See `get_eelt_segments()`

flattoflat [float] [m] Default : 1.256

gap [float] [m] Default : 0.004

secondary_radius [float] [m] Default : 5

n_supports [int] Default : 6

support_width [float] [m] Default : 0.2

support_angle_offset [float] [deg] Default : 0

n_missing [int] Default : None. Number of segments missing

pupil_inner_radius [float] [m] Default : None # Plan A: 5.6m, Plan B: 11.5m

pupil_outer_radius [float] [m] Default : 19

See also:

`get_eelt_segments()`

seeing_psf

`simcado.psf.`**`seeing_psf`**(*fwhm=0.8*, *psf_type='moffat'*, *size=1024*, *pix_res=0.004*, *filename=None*)

Return a seeing limited PSF

Parameters

fwhm [float, optional] [arcsec] Default = 0.8

psf_type [str, optional] ["moffat, "gaussian"] Default = "moffat"

size [int, optional] [pixel] Default = 1024

pix_res [float, optional] [arcsec] Default = 0.004

filename [str, optional] Default = None. If filename is not None, the resulting FITS object will be saved to disk

Returns

seeing_psf [2D-array]

Notes

# Moffat description # https://www.gnu.org/software/gnuastro/manual/html_node/PSF.html # # Approximate parameters - Bendinelli 1988 # beta = 4.765 - Trujillo et al. 2001

get_eelt_segments

`simcado.psf.`**`get_eelt_segments`**(*plan='A'*, *missing=None*, *return_missing_segs=False*, *inner_diam=10.6*, *outer_diam=39.0*)

Generate a list of segments for POPPY for the E-ELT

Parameters

plan [str, optional] ["A", "B"], Default = "A" * Plan A is for a fully populated mirror (798 segments) * Plan B has the inner 5 rings missing (588 segments) and a further 5 random segments missing (583 segments)

missing [int, list, optional] Default = None. If an integer is passed, this many random segments are removed. If `missing` is a list, entries refer to specific segment IDs

return_missing_segs [bool, optional] Defualt is False. Returns the missing segment numbers

inner_diam [float, optional] [m] Default = 10.6. Diameter which produces ESO's mirror configuration

outer_diam [float, optional] [m] Default = 39.0. Diameter which produces ESO's mirror configuration

Returns

segs [list] A list of segment IDs for the mirror segments.

missing [list, conditional] Only returned if `return_missing_segs == True`. A list of segment IDS for the segments which are missing.

make_foreign_PSF_cube

`simcado.psf.`**`make_foreign_PSF_cube`**(*fnames*, *out_name=None*, *window=None*, *pix_res_orig=None*, *pix_res_final=None*, *wavelengths=None*)

Combine several PSF FITS images into a single PSF FITS file

Parameters

fnames [list] List of path names to the FITS files

out_name [str, optional] If out_name is not `None`, the resulting FITS file is saved under the name `out_name`

window [int, list, tuple, optional] If window is not `None`, a windowed section of the PSFs are extracted window = (left, right, top, bottom) window = square radius

Examples

```
>>> from glob import glob
>>> import simcado as sim
>>> fnames = glob("D:\Share_VW\Data_for_SimCADO\PSFs\yann_2016_11_10\*.fits")
>>> sim.psf.make_foreign_PSF_cube(fnames, "PSF_SCAO.fits",
                                  window=512,
                                  pix_res_orig=[0.0028, 0.0037, 0.00492],
                                  pix_res_final=[0.004, 0.004, 0.004],
                                  wavelengths=[1.25,1.65,2.2])
```

Classes

`PSF`(size, pix_res)	Point spread function (single layer) base class
`PSFCube`(lam_bin_centers)	Class holding wavelength dependent point spread function.
`MoffatPSF`(fwhm, **kwargs)	Generate a PSF for a Moffat function.
`MoffatPSFCube`(lam_bin_centers[, fwhm])	Generate a list of MoffatPSFs for wavelengths defined in lam_bin_centers
`AiryPSF`(fwhm[, obscuration, size, pix_res])	Generate a PSF for an Airy function with an equivalent FWHM
`AiryPSFCube`(lam_bin_centers[, fwhm])	Generate a list of AiryPSFs for wavelengths defined in lam_bin_centers
`GaussianPSF`(fwhm, **kwargs)	Generate a PSF for an Gaussian function
`GaussianPSFCube`(lam_bin_centers[, fwhm])	Generate a list of GaussianPSFs for wavelengths defined in lam_bin_centers
`DeltaPSF`(**kwargs)	Generate a PSF with a delta function at position (x,y)

Continued on next page

Table 15 – continued from previous page

`DeltaPSFCube`(lam_bin_centers[, positions])	Generate a list of DeltaPSFs for wavelengths defined in lam_bin_centers
`CombinedPSF`(psf_list, **kwargs)	Generate a PSF from a collection of several PSFs.
`CombinedPSFCube`(psf_list, **kwargs)	Generate a list of CombinedPSFCubes from the list of psfs in psf_list
`UserPSF`(filename, **kwargs)	Import a PSF from a FITS file.
`UserPSFCube`(filename, lam_bin_centers)	Read in a psf previously saved as a FITS file
`FieldVaryingPSF`(**kwargs)	

PSF

class simcado.psf.**PSF**(*size*, *pix_res*)

Bases: `object`

Point spread function (single layer) base class

Parameters

size [int] [pixel] the side length of the array

pix_res [float] [arcsec] the pixel scale of the array

Methods Summary

`convolve`(self, kernel)	Convolve the PSF with another kernel.
`resample`(self, new_pix_res)	Resample the PSF array onto a new grid
`resize`(self, new_size)	Resize the PSF.
`set_array`(self, array[, threshold])	Set the spatial flux distribution array for the PSF

Methods Documentation

convolve(*self*, *kernel*)

Convolve the PSF with another kernel. The PSF keeps its shape

Parameters

kernel [np.array, PSF] Either a numpy.ndarray or a PSF (sub)class

Returns

psf_new [PSF]

resample(*self*, *new_pix_res*)

Resample the PSF array onto a new grid

Not perfect, but conserves flux

Parameters

new_pix_res [float] [arcsec] the pixel resolution of the returned array

Returns

psf_new [PSF]

Examples

```
>>> new_PSF = old_PSF.resample(new_pix_res)
```

resize(*self*, *new_size*)
Resize the PSF. The target shape is (new_size, new_size).

Parameters

new_size [int] the new square dimensions of the PSF array in pixels

set_array(*self*, *array*, *threshold=1e-15*)
Set the spatial flux distribution array for the PSF

Renormalise the array make sure there aren't any negative values that will screw up the flux

Parameters

array [np.ndarray] the array representing the PSF

threshold [float] by default set to 1E-15. Below this, the array is set to 0

PSFCube

class simcado.psf.**PSFCube**(*lam_bin_centers*)
Bases: object

Class holding wavelength dependent point spread function.

Parameters

lam_bin_centers [array] [um] the centre of each wavelength slice

Notes

- len(self) return the number of layers in the psf
- self[i] returns the PSF object for layer i. If the __array__ function is called, self[i] will return the array associated with the instance e.g plt.imshow(self[i]) will plot PSF.array from self.psf_slices[i]
- Maths operators *,+,- act equally on all PSF.arrays in self.psf_slices

Methods Summary

convolve(self, kernel_list)	Convolve a list of PSFs with a list of kernels
export_to_fits(self, filename[, clobber])	Export the psf to a FITS file for later use
nearest(self, lam)	Returns the PSF closest to the desired wavelength, lam [um]
resample(self, new_pix_res)	Resample the the list of PSF array onto a new grid
resize(self, new_size)	Resize the list of PSFs.

Methods Documentation

convolve(*self*, *kernel_list*)
Convolve a list of PSFs with a list of kernels

Parameters

kernel_list [list] list of PSF objects of 2D arrays

export_to_fits (*self*, *filename*, *clobber=True*)
Export the psf to a FITS file for later use

Parameters

filename [str]

nearest (*self*, *lam*)
Returns the PSF closest to the desired wavelength, lam [um]

Parameters

lam [float] [um] desired wavelength

Returns

psf_slice [PSF]

resample (*self*, *new_pix_res*)
Resample the the list of PSF array onto a new grid

Not perfect, but conserves flux

Parameters

new_pix_res [float] [arcsec] the pixel resolution of the returned array

Returns

psf_new [PSF]

Examples

```
>>> new_PSF = old_PSF.resample(new_pix_res)
```

resize (*self*, *new_size*)
Resize the list of PSFs. The target shape is (new_size, new_size).

Parameters

new_size [int] [pixel] the new size of the PSF array in pixels

MoffatPSF

class simcado.psf.**MoffatPSF** (*fwhm*, ***kwargs*)
Bases: simcado.psf.PSF

Generate a PSF for a Moffat function. Alpha is generated from the FWHM and Beta = 4.765 (from Trujillo et al. 2001)

Parameters

fwhm [float] [arcsec] the equivalent FWHM of the Airy disk core.

size [int] [pixel] the side length of the array

pix_res [float] [arcsec] the pixel scale used in the array, default is 0.004

mode [str] ['oversample','linear_interp'] see Kernel2D (scipy.convolution.core)

MoffatPSFCube

class simcado.psf.**MoffatPSFCube**(*lam_bin_centers*, *fwhm=None*, ***kwargs*)

Bases: `simcado.psf.PSFCube`

Generate a list of MoffatPSFs for wavelengths defined in lam_bin_centers

Parameters

lam_bin_centers [array, list] [um] a list with the centres of each wavelength slice

fwhm [array, list] [arcsec] the equivalent FWHM of the PSF.

diameter [float] [m] diamter of primary mirror. Default is 39.3m.

mode [str] ['oversample','linear_interp'] see Kernel2D (scipy.convolution.core)

AiryPSF

class simcado.psf.**AiryPSF**(*fwhm*, *obscuration=0.0*, *size=255*, *pix_res=0.004*, ***kwargs*)

Bases: `simcado.psf.PSF`

Generate a PSF for an Airy function with an equivalent FWHM

Parameters

fwhm [float] [arcsec] the equivalent FWHM of the Airy disk core.

size [int] [pixel] the side length of the array

pix_res [float] [arcsec] the pixel scale used in the array, default is 0.004

obscuration [float] [0..1] radius of inner obscuration as fraction of aperture radius

mode [str] ['oversample','linear_interp'] see Kernel2D (scipy.convolution.core)

AiryPSFCube

class simcado.psf.**AiryPSFCube**(*lam_bin_centers*, *fwhm=None*, ***kwargs*)

Bases: `simcado.psf.PSFCube`

Generate a list of AiryPSFs for wavelengths defined in lam_bin_centers

Parameters

lam_bin_centers [array, list] [um] a list with the centres of each wavelength slice

fwhm [array, list] [arcsec] the equivalent FWHM of the PSF.

diameter [float] [m] diamter of primary mirror. Default is 39.3m.

mode [str] ['oversample','linear_interp'] see Kernel2D (scipy.convolution.core)

GaussianPSF

class simcado.psf.**GaussianPSF**(*fwhm*, ***kwargs*)

Bases: `simcado.psf.PSF`

Generate a PSF for an Gaussian function

Parameters

fwhm [float] [arcsec] the equivalent FWHM of the Airy disk core.

size [int] [pixel] the side length of the array

pix_res [float] [arcsec] the pixel scale used in the array, default is 0.004

mode [str] ['oversample','linear_interp'] see Kernel2D (scipy.convolution.core)

GaussianPSFCube

class `simcado.psf.`**`GaussianPSFCube`**(*lam_bin_centers*, *fwhm=None*, ***kwargs*)

Bases: `simcado.psf.PSFCube`

Generate a list of GaussianPSFs for wavelengths defined in lam_bin_centers

Parameters

lam_bin_centers [array, list] [um] a list with the centres of each wavelength slice

fwhm [array, list] [arcsec] the equivalent FWHM of the PSF.

diameter [float] [m] diamter of primary mirror. Default is 39.3m.

mode [str] ['oversample','linear_interp'] see Kernel2D (scipy.convolution.core)

DeltaPSF

class `simcado.psf.`**`DeltaPSF`**(***kwargs*)

Bases: `simcado.psf.PSF`

Generate a PSF with a delta function at position (x,y)

Parameters

position [tuple] [pixel] where (x,y) on the array is where the delta function goes default is (x,y) = (0,0) and is the centre of the array

size [int] [pixel] the side length of the array

pix_res [float] [arcsec] the pixel scale used in the array, default is 0.004

DeltaPSFCube

class `simcado.psf.`**`DeltaPSFCube`**(*lam_bin_centers*, *positions=(0, 0)*, ***kwargs*)

Bases: `simcado.psf.PSFCube`

Generate a list of DeltaPSFs for wavelengths defined in lam_bin_centers

Parameters

lam_bin_centers [array] [um] the centre of each wavelength slice

positions [tuple, list]

(x,y) either a tuple, or a list of tuples denoting the position of the delta function

pix_res [float] [arcsec], pixel scale of the PSF. Default is 0.004 arcsec

size [int] [pixel] side length of the PSF array

CombinedPSF

class `simcado.psf.`**`CombinedPSF`**(*psf_list*, ***kwargs*)

Bases: `simcado.psf.PSF`

Generate a PSF from a collection of several PSFs.

Parameters

psf_list [list] A list of PSF objects

size [int] [pixel] the side length in pixels of the array

CombinedPSFCube

class `simcado.psf.`**`CombinedPSFCube`**(*psf_list*, ***kwargs*)

Bases: `simcado.psf.PSFCube`

Generate a list of CombinedPSFCubes from the list of psfs in psf_list

Parameters

lam_bin_centers [array, list] [um] a list with the centres of each wavelength slice

fwhm [array, list] [arcsec] the equivalent FWHM of the PSF.

diameter [float] [m] diamter of primary mirror. Default is 39.3m.

UserPSF

class `simcado.psf.`**`UserPSF`**(*filename*, ***kwargs*)

Bases: `simcado.psf.PSF`

Import a PSF from a FITS file.

Parameters

filename [str] path to the FITS file to be read in

fits_ext [int, optional] the FITS extension number (default 0) for the data

pix_res [float, optional] [arcsec] the pixel scale used in the array, default is 0.004

UserPSFCube

class `simcado.psf.`**`UserPSFCube`**(*filename*, *lam_bin_centers*)

Bases: `simcado.psf.PSFCube`

Read in a psf previously saved as a FITS file

Keywords needed for a psf to be read in: NSLICES, WAVECENT, NAXIS1, CDELT1, PSF_TYPE, DESCRIPT

Parameters

filename [str] the path to the FITS file holding the cube

See also:

`make_foreign_PSF_cube()`

Notes

A separate function will exist to convert foreign PSF FITS files into `simcado.psf` readable FITS files

5.13.6 `simcado.simulation module`

simcado.simulation Module

simulation.py

Functions

`run`(src[, mode, cmds, opt_train, fpa, ...])	Run a MICADO simulation with default parameters
`snr`(exptimes, mags[, filter_name, cmds])	Returns the signal-to-noise ratio(s) for given exposure times and magnitudes
`check_chip_positions`([filename, x_cen, ...])	Creates a series of grids of stars and generates the output images
`limiting_mags`([exptimes, filter_names, ...])	Return or plot a graph of the limiting magnitudes for MICADO
`zeropoint`([filter_name])	Returns the zero point magnitude for a SimCADO filter

run

`simcado.simulation.`**`run`**(*src*, *mode='wide'*, *cmds=None*, *opt_train=None*, *fpa=None*, *detector_layout='small'*, *filename=None*, *return_internals=False*, *filter_name=None*, *exptime=None*, *sub_pixel=False*, *sim_data_dir=None*, ***kwargs*)

Run a MICADO simulation with default parameters

Parameters

src [simcado.Source] The object of interest

mode [str, optional] ["wide", "zoom"] Default is "wide", for a 4mas FoV. "Zoom" -> 1.5mas

cmds [simcado.UserCommands, optional] A custom set of commands for the simulation. Default is None

opt_train [simcado.OpticalTrain, optional] A custom optical train for the simulation. Default is None

fpa [simcado.Detector, optional] A custom detector layout for the simulation. Default is None

detector_layout [str, optional] ["small", "centre", "full", "tiny"] Default is "small". "small" - 1x 1k-detector centred in the FoV "tiny" - 128 x 128 pixels centred in the FoV "centre" - 1x 4k-detector centred in the FoV "full" - 9x 4k-detectors

filename [str, optional] The filepath for where the FITS images should be saved. Default is None. If None, the output images are returned to the user as FITS format astropy.io.HDUList objects.

return_internals [bool] [False, True] Default is False. If True, the `UserCommands`, `OpticalTrain` and `Detector` objects used in the simulation are returned in a tuple: `return hdu, (cmds, opt_train, fpa)`

filter_name [str, TransmissionCurve] Analogous to passing INST_FILTER_TC as a keyword argument

exptime [int, float]

[s] Total integration time. Currently, this is observed in one DIT (i.e. NDIT=1). Use OBS_DIT and OBS_NDIT for more general setup.

sim_data_dir [str] Path to where the data is kept

snr

`simcado.simulation.`**`snr`**(*exptimes*, *mags*, *filter_name='Ks'*, *cmds=None*, ***kwargs*)

Returns the signal-to-noise ratio(s) for given exposure times and magnitudes

Each time this runs, simcado runs a full simulation on a grid of stars. Therefore if you are interested in the SNR for many difference expoure times and a range of magnitudes, it is faster to pass all of them at once to this function. See the example section below.

Parameters

exptimes [float, list] [s] A single or multiple exposure times

mags [float, list] [mag] A single or multiple magnitudes

filter_name [str, optional] The name of the filter to be used - See `get_filter_set()` The default is "Ks"

cmds [UserCommands object, optional] Extra commands to be passed to `simcado.simulation.run()`.

Returns

snr_return [list] A list of SNR values for each exposure time and each magnitude

Examples

A basic example of wanting the SNR for a Ks=24 star in a 1 hr observation

```
>>> snr(exptimes=3600, mags=24)
[72.69760133863036]
```

However this is slow because it runs a full simulation. Hence it is better to do more at once If we want the SNR for the range of magnitudes J=[15, 20, 25, 30] for a 1 hr observation:

```
>>> snr(exptimes=3600, mags=[15,20,25,30], filter_name="J")
[array([  2.35125027e+04,   2.74921916e+03,   8.97552604e+01,
  8.18183097e-01])]
```

Now if we were interested in different exposure times, say 10 minutes and 5 hours, for a 24th magnitude star in the narrow band Br$gamma$ filter:

```
>>> # Chekc the name of the Brackett Gamma filter
>>> [name for name in simcado.optics.get_filter_set() if "Br" in name]
['BrGamma']
>>> snr(exptimes=[600, 18000], mags=24, filter_name="BrGamma")
[8.016218764390803, 42.71569256185457]
```

check_chip_positions

`simcado.simulation.`**`check_chip_positions`**(*filename='src.fits', x_cen=17.084, y_cen=17.084, n=0.3, mode='wide'*)

Creates a series of grids of stars and generates the output images

THe number of stars in each grid corresponds to the id number of the chip

limiting_mags

`simcado.simulation.`**`limiting_mags`**(*exptimes=None, filter_names=None, AB_corrs=None, limiting_sigma=5, return_mags=True, make_graph=False, mmin=22, mmax=31, cmds=None, **kwargs*)

Return or plot a graph of the limiting magnitudes for MICADO

Parameters

exptimes [array] [s] Exposure times for which limiting magnitudes should be found

filter_names [list] A list of filters. See `simcado.optics.get_filter_set()`

AB_corrs [list] [mag] A list of magnitude corrections to convert from Vega to AB magnitudes

limiting_sigma [float] [sigma] The number of sigmas to use to define the limiting magnitude. Default is 5*sigma

return_mags [bool] If True (defualt), the limiting magnitude are returned

make_graph [bool] If True (defualt), a graph of the limiting magnitudes vs exposure time is plotted Calls `plot_exptime_vs_limiting_mag()`

cmds [simcado.UserCommands] A custom set of commands for building the optical train

Returns

mags_all [list] [mag] If `return_mags=True`, returns a list of limiting magnitudes for each exposure time for each filter Dimensions are [n, m] where n is the number of filters and m is the number of exposure times passed

Notes

Vega to AB = {“J” : 0.91 , “H” : 1.39 , “Ks” : 1.85}

Examples

:

```
>>> # Set 30 logarithmic time bins between 1 sec and 5 hours
>>> exptimes = np.logspace(0, np.log10(18000), num=30, endpoint=True)
>>> limiting_mags(exptimes=exptimes, filter_names=["J", "PaBeta"],
...                make_graph=False)
```

zeropoint

`simcado.simulation.`**`zeropoint`**(*filter_name='TC_filter_Ks.dat'*)

Returns the zero point magnitude for a SimCADO filter

This is an end-to-end simulation which aims to take into account all transmission effects incorporated in a SimCADO simulation.

The returned zeropoint is for an exposure of 1s. Therefore, magnitudes from measured fluxes in simulated images should be calculated as following

mag = -2.5*np.log10(counts/texp) + zp

where counts are the background subtracted counts, texp is the exposure time and zp is the zeropoint for the filter in question, calculated here.

Parameters

filter_name: A SimCADO filter

Returns

zp: the zeropoint magnitude

5.13.7 `simcado.source` module

simcado.source Module

The module that contains the functionality to create Source objects

Module Summary

The Source is essentially a list of spectra and a list of positions. The list of positions contains a reference to the relevant spectra. The advantage here is that if there are repeated spectra in a data cube, we can reduce the amount of calculations needed. Furthermore, if the input is originally a list of stars, etc., where the position of a star is not always an integer multiple of the plate scale, we can keep the information until the PSFs are needed.

Classes

Source

Functions

Functions to create `Source` objects

```
empty_sky()
star(mag, filter_name="Ks", spec_type="A0V", x=0, y=0)
stars(mags, filter_name="Ks", spec_types=["A0V"], x=[0], y=[0])
star_grid(n, mag_min, mag_max, filter_name="Ks", separation=1, area=1,
          spec_type="A0V")
source_from_image(images, lam, spectra, pix_res, oversample=1,
                  units="ph/s/m2", flux_threshold=0,
                  center_pixel_offset=(0, 0))
source_1E4_Msun_cluster(distance=50000, half_light_radius=1)
```

Functions for manipulating spectra for a `Source` object

```
scale_spectrum(lam, spec, mag, filter_name="Ks", return_ec=False)
scale_spectrum_sb(lam, spec, mag_per_arcsec, pix_res=0.004,
                  filter_name="Ks", return_ec=False)
flat_spectrum(mag, filter_name="Ks", return_ec=False)
flat_spectrum_sb(mag_per_arcsec, filter_name="Ks", pix_res=0.004,
                 return_ec=False)
```

Functions regarding photon flux and magnitudes

```
zero_magnitude_photon_flux(filter_name)
_get_stellar_properties(spec_type, cat=None, verbose=False)
_get_stellar_mass(spec_type)
_get_stellar_Mv(spec_type)
_get_pickles_curve(spec_type, cat=None, verbose=False)
```

Helper functions

```
value_at_lambda(lam_i, lam, val, return_index=False)
SED(spec_type, filter_name="V", magnitude=0.)
```

Functions

`star`([spec_type, mag, filter_name, x, y])	Creates a simcado.Source object for a star with a given magnitude
`stars`([spec_types, mags, filter_name, x, y])	Creates a simcado.Source object for a bunch of stars.
`cluster`([mass, distance, half_light_radius])	Generate a source object for a cluster
`point_source`([spectrum, mag, filter_name, x, y])	Creates a simcado.Source object for a point source with a given magnitude
`spiral`(half_light_radius, plate_scale[, …])	Create a extended `Source` object for a “Galaxy”
`spiral_profile`(r_eff[, ellipticity, angle, …])	Creates a spiral profile with arbitary parameters
`elliptical`(half_light_radius, plate_scale[, …])	Create a extended `Source` object for a “Galaxy”
`sersic_profile`([r_eff, n, ellipticity, …])	Returns a 2D array with a normalised Sersic profile
`source_from_image`(images, lam, spectra, …)	Create a Source object from an image or a list of images.
`star_grid`(n, mag_min, mag_max[, …])	Creates a square grid of A0V stars at equal magnitude intervals
`empty_sky`()	Returns an empty source so that instrumental fluxes can be simulated
`SED`(spec_type[, filter_name, magnitude])	Return a scaled SED for a star or type of galaxy
`redshift_SED`(z, spectrum, mag[, filter_name])	Redshift a SimCADO SED and scale it to a magnitude in a user specified filter
`sie_grad`(x, y, par)	Compute the deflection of an SIE (singular isothermal ellipsoid) potential
`apply_grav_lens`(image[, x_cen, y_cen, …])	Apply a singular isothermal ellipsoid (SIE) gravitational lens to an image
`get_SED_names`([path])	Return a list of the SEDs installed in the package directory
`scale_spectrum`(lam, spec, mag[, …])	Scale a spectrum to be a certain magnitude
`scale_spectrum_sb`(lam, spec, mag_per_arcsec)	Scale a spectrum to be a certain magnitude per arcsec2
`flat_spectrum`(mag[, filter_name, return_ec])	Return a flat spectrum scaled to a certain magnitude
`flat_spectrum_sb`(mag_per_arcsec[, …])	Return a flat spectrum for a certain magnitude per arcsec
`value_at_lambda`(lam_i, lam, val[, return_index])	Return the value at a certain wavelength - i.e.

Continued on next page

Table 19 – continued from previous page

`BV_to_spec_type`(B_V)	Returns the latest main sequence spectral type(s) for (a) B-V colour
`zero_magnitude_photon_flux`(filter_name)	Return the number of photons for a m=0 star for a filter with synphot
`mag_to_photons`(filter_name[, magnitude])	Return the number of photons for a certain filter and magnitude
`photons_to_mag`(filter_name[, photons])	Return the number of photons for a certain filter and magnitude
`_get_pickles_curve`(spec_type[, cat, verbose])	Returns the emission curve for a single or list of `spec_type`, normalised to 5556A
`_get_stellar_properties`(spec_type[, cat, ...])	Returns an `astropy.Table` with the list of properties for the star(s) in `spec_type`
`_get_stellar_Mv`(spec_type)	Returns a single (or list of) float(s) with the V-band absolute magnitude(s)
`_get_stellar_mass`(spec_type)	Returns a single (or list of) float(s) with the stellar mass(es)

star

`simcado.source.`**`star`**(*spec_type='A0V'*, *mag=0*, *filter_name='Ks'*, *x=0*, *y=0*, ***kwargs*)

Creates a simcado.Source object for a star with a given magnitude

This is just the single star variant for `simcado.source.stars()`

Parameters

spec_type [str] the spectral type of the star, e.g. "A0V", "G5III"

mag [float] magnitude of star

filter_name [str] Filter in which the magnitude is given. Can be the name of any filter curve file in the simcado/data folder, or a path to a custom ASCII file

x, y [float, int, optional] [arcsec] the x,y position of the star on the focal plane

Returns

source [`simcado.Source`]

See also:

`stars`

stars

`simcado.source.`**`stars`**(*spec_types='A0V'*, *mags=0*, *filter_name='Ks'*, *x=None*, *y=None*, ***kwargs*)

Creates a simcado.Source object for a bunch of stars.

Parameters

spec_types [str, list of strings] the spectral type(s) of the stars, e.g. "A0V", "G5III" Default is "A0V"

mags [float, array] [mag] magnitudes of the stars.

filter_name [str,] Filter in which the magnitude is given. Can be the name of any filter curve file in the simcado/data folder, or a path to a custom ASCII file

x, y [arrays] [arcsec] x and y coordinates of the stars on the focal plane

Returns

source [`simcado.Source`]

Examples

Create a `Source` object for a random group of stars

```
>>> import numpy as np
>>> from simcado.source import stars
>>>
>>> spec_types = ["A0V", "G2V", "K0III", "M5III", "O8I"]
>>> ids = np.random.randint(0,5, size=100)
>>> star_list = [spec_types[i] for i in ids]
>>> mags = np.random.normal(20, 3, size=100)
>>>
>>> src = stars(spec_types, mags, filter_name="Ks")
```

If we don't specify any coordinates all stars have the position (0, 0). **All positions are in arcsec.** There are two possible ways to add positions. If we know them to begin with we can add them when generating the source full of stars

```
>>> x, y = np.random.random(-20, 20, size=(100,2)).tolist()
>>> src = stars(star_list, mags, filter_name="Ks", x=x, y=y)
```

Or we can add them to the `Source` object directly (although, there are less checks to make sure the dimensions match here):

```
>>> src.x, src.y = x, y
```

cluster

`simcado.source.`**`cluster`**(*mass=1000.0*, *distance=50000*, *half_light_radius=1*)

Generate a source object for a cluster

The cluster distribution follows a gaussian profile with the `half_light_radius` corresponding to the HWHM of the distribution. The choice of stars follows a Kroupa IMF, with no evolved stars in the mix. Ergo this is more suitable for a young cluster than an evolved custer

Parameters

mass [float] [Msun] Mass of the cluster (not number of stars). Max = 1E5 Msun

distance [float] [pc] distance to the cluster

half_light_radius [float] [pc] half light radius of the cluster

Returns

src [simcado.Source]

Examples

Create a `Source` object for a young open cluster with half light radius of around 0.2 pc at the galactic centre and 100 solar masses worth of stars:

```
>>> from simcado.source import cluster
>>> src = cluster(mass=100, distance=8500, half_light_radius=0.2)
```

point_source

`simcado.source.point_source`(*spectrum='AOV', mag=0, filter_name='TC_filter_Ks.dat', x=0, y=0, **kwargs*)

Creates a simcado.Source object for a point source with a given magnitude

This is a variant for `simcado.source.star()` but can accept any SED as well as an user provided EmissionCurve object

Parameters

spectrum [str, EmissionCurve, optional] The spectrum to be associated with the point source. Values can either be: - the name of a SimCADO SED spectrum : see get_SED_names() - an EmissionCurve with a user defined spectrum

mag [float] magnitude of object

filter_name [filter_name] Default is "Ks". Values can be either: - the name of a SimCADO filter : see optics.get_filter_set() - or a TransmissionCurve containing a user-defined filter

x, y [float, int, optional] [arcsec] the x,y position of the point source on the focal plane

Returns

source [`simcado.Source`]

See also:

`star`

spiral

`simcado.source.spiral`(*half_light_radius, plate_scale, magnitude=10, filter_name='Ks', normalization='total', spectrum='spiral', **kwargs*)

Create a extended `Source` object for a "Galaxy"

Parameters

half_light_radius [float] [arcsec]

plate_scale [float] [arcsec]

magnitude [float] [mag, mag/arcsec2]

filter_name [str, TransmissionCurve, optional. Default is "Ks". Values can be either:]

- the name of a SimCADO filter : see optics.get_filter_set()
- or a TransmissionCurve containing a user-defined filter

normalization [str, optional] ["half-light", "centre", "total"] Where in the profile equals unityy If normalization equals:

- "half-light" : the pixels at the half-light radius have a surface brightness of `magnitude` [mag/arcsec2]
- "centre" : the maximum pixels have a surface brightness of `magnitude` [mag/arcsec2]

- "total" : the whole image has a brightness of `magnitude` [mag]

spectrum [str, EmissionCurve, optional] The spectrum to be associated with the galaxy. Values can either be: - the name of a SimCADO SED spectrum : see get_SED_names() - an EmissionCurve with a user defined spectrum

Returns

galaxy_src [simcado.Source]

See also:

`sersic_profile`

`spiral_profile`

`optics.get_filter_set`

`source.get_SED_names`

`spectral.TransmissionCurve`

`spectral.EmissionCurve`

spiral_profile

`simcado.source.`**`spiral_profile`**(*r_eff*, *ellipticity=0.5*, *angle=45*, *n_arms=2*, *tightness=4.0*, *arms_width=0.1*, *central_brightness=10*, *normalization='total'*, *width=1024*, *height=1024*, *oversample=1*, ***kwargs*)

Creates a spiral profile with arbitary parameters

Parameters

r_eff [float] [pixel] Effective (half-light) radius

ellipticity [float] Ellipticity is defined as (a - b)/a. Default = 0.5

angle [float] [deg] Default = 45. Rotation anti-clockwise from the x-axis

n_arms [int] Number of spiral arms

tightness [float] How many times an arm crosses the major axis. Default = 4.

arms_width [float] An arbitary scaling factor for how think the arms should be. Seems to scale with central_brightness. Default = 0.1

central_brightness [float] An arbitary scaling factor for the strength of the central region. Has some connection to ars_width. Default = 10

normalization [str, optional] ["half-light", "centre", "total"] Where the profile equals unity If normalization equals: - "centre" : the maximum values are set to 1 - "total" : the image sums to 1

width, height [int, int] [pixel] Dimensions of the image

x_offset, y_offset [float] [pixel] The distance between the centre of the profile and the centre of the image

oversample [int] Factor of oversampling, default factor = 1. If > 1, the model is discretized by taking the average of an oversampled grid.

Returns

img [np.ndarray] A 2D image of a spiral disk

See also:

`sersic_profile`

Notes

The intensity drop-off is dictated by a sersic profile of with indes n=1, i.e. an exponential drop-off. This can be altered by passing the keyword "n=" as an optional parameter.

Spiral structure taken from here: https://stackoverflow.com/questions/36095775/creating-a-spiral-structure-in-python-using-hyperbolic-tangent

elliptical

`simcado.source.`**`elliptical`**(*half_light_radius*, *plate_scale*, *magnitude=10*, *n=4*, *filter_name='Ks'*, *normalization='total'*, *spectrum='elliptical'*, ***kwargs*)

Create a extended `Source` object for a "Galaxy"

Parameters

half_light_radius [float] [arcsec]

plate_scale [float] [arcsec]

magnitude [float] [mag, mag/arcsec2]

n [float, optional] Power law index. Default = 4 - n=1 for exponential (spiral), - n=4 for de Vaucouleurs (elliptical)

filter_name [str, TransmissionCurve, optional] Default is "Ks". Values can be either: - the name of a SimCADO filter : see optics.get_filter_set() - or a TransmissionCurve containing a user-defined filter

normalization [str, optional] ["half-light", "centre", "total"] Where the profile equals unity If normalization equals:

- "half-light" : the pixels at the half-light radius have a surface brightness of `magnitude` [mag/arcsec2]
- "centre" : the maximum pixels have a surface brightness of `magnitude` [mag/arcsec2]
- "total" : the whole image has a brightness of `magnitude` [mag]

spectrum [str, EmissionCurve, optional] The spectrum to be associated with the galaxy. Values can either be: - the name of a SimCADO SED spectrum : see get_SED_names() - an EmissionCurve with a user defined spectrum

Returns

galaxy_src [`Source`]

See also:

`simcado.source.sersic_profile()`

`simcado.optics.get_filter_set()`

`simcado.source.get_SED_names()`

`simcado.spectral.TransmissionCurve`

`simcado.spectral.EmissionCurve`

sersic_profile

simcado.source.**sersic_profile**(*r_eff=100*, *n=4*, *ellipticity=0.5*, *angle=30*, *normalization='total'*, *width=1024*, *height=1024*, *x_offset=0*, *y_offset=0*, *oversample=1*)

Returns a 2D array with a normalised Sersic profile

Parameters

r_eff [float] [pixel] Effective (half-light) radius

n [float] Power law index. - n=1 for exponential (spiral), - n=4 for de Vaucouleurs (elliptical)

ellipticity [float] Ellipticity is defined as (a - b)/a. Default = 0.5

angle [float] [deg] Default = 30. Rotation anti-clockwise from the x-axis

normalization [str, optional] ["half-light", "centre", "total"] Where the profile equals unity If normalization equals: - "half-light" : the pixels at the half-light radius are set to 1 - "centre" : the maximum values are set to 1 - "total" : the image sums to 1

width, height [int] [pixel] Dimensions of the image

x_offset, y_offset [float] [pixel] The distance between the centre of the profile and the centre of the image

oversample [int] Factor of oversampling, default factor = 1. If > 1, the model is discretized by taking the average of an oversampled grid.

Returns

img [2D array]

Notes

Most units are in [pixel] in this function. This differs from `galaxy()` where parameter units are in [arcsec] or [pc]

source_from_image

simcado.source.**source_from_image**(*images*, *lam*, *spectra*, *plate_scale*, *oversample=1*, *units='ph/s/m2'*, *flux_threshold=0*, *center_offset=(0, 0)*, *conserve_flux=True*, ***kwargs*)

Create a Source object from an image or a list of images.

Note: `plate_scale` is the original plate scale of the images. If this is not the same as the plate scale of the `Detector` then you will need to specify oversample to interpolate between the two scales. I.e. oversample = Image plate scale / Detector plate scale

Parameters

images [np.ndarray, list] A single or list of np.ndarrays describing where the flux is coming from The spectrum for each pixel in the image is weighted by the pixel value.

lam [np.ndarray] An array contains the centres of the wavelength bins for the spectra

spectra [np.ndarray] A (n,m) array with n spectra, each with m bins

plate_scale [float] [arcsec] The plate scale of the images in arcseconds (e.g. 0.004"/pixel)

oversample [int] The factor with which to oversample the image. Each image pixel is split into (oversample)^2 individual point sources.

units [str, optional] The energy units of the spectra. Default is [ph/s/m2]

flux_threshold [float, optional] If there is noise in the image, set threshold to the noise limit so that only real photon sources are extracted. Default is 0.

center_offset [(float, float)] [arcsec] If the centre of the image is offset, add this offset to (x,y) coordinates.

conserve_flux [bool, optional] If True, when the image is rescaled, flux is conserved i.e. np.sum(image) remains constant If False, the maximum value of the image stays constant after rescaling i.e. np.max(image) remains constant

Returns

src [`Source` object]

Notes

Currently only one object per image is supported.

Examples

To create a `Source` object we need an image that describes the spatial distribution of the object of interest and spectrum. For the sake of ease we will assign a generic elliptical galaxy spectrum to the image.:

```
>>> from astropy.io import fits
>>> from simcado.source import SED, source_from_image
>>>
>>> im = fits.getdata("galaxy.fits")
>>> lam, spec = SED("elliptical")
>>> src = source_from_image(im, lam, spec,
                            plate_scale=0.004)
```

Note Here we have assumed that the plate scale of the image is the same as the MICADO wide-field mode, i.e. 0.004 arcseconds. If the image is from a real observation, or it was generated with a different pixel scale, we will need to tell SimCADO about this:

```
>>> src = source_from_image(im, lam, spec,
                            plate_scale=0.01,
                            oversample=2.5)
```

If the image is from real observations, chances are good that the background flux is higher than zero. We can set a `threshold` in order to tell SimCADO to ignore all pixel with values below the background level:

```
>>> src = source_from_image(im, lam, spec,
                            plate_scale=0.01,
                            oversample=2.5,
                            flux_threshold=0.2)
```

Finally, if the image centre is not the centre of the observation, we can shift the image relative to the MICADO field of view. The units for the offset are [arcsec]:

```
>>> src = source_from_image(im, lam, spec,
                            plate_scale=0.01,
                            oversample=2.5,
                            flux_threshold=0.2,
                            center_offset=(10,-15))
```

star_grid

`simcado.source.`**`star_grid`**`(`*n, mag_min, mag_max, filter_name='Ks', separation=1, spec_type='A0V'*`)`

Creates a square grid of A0V stars at equal magnitude intervals

Parameters

n [float] the number of stars in the grid

mag_min, mag_max [float] [vega mag] the minimum (brightest) and maximum (faintest) magnitudes for stars in the grid

filter_name [str] any filter that is in the SimCADO package directory. See `simcado.optics.get_filter_set()`

separation [float, optional] [arcsec] an average speration between the stars in the grid can be specified. Default is 1 arcsec

spec_type [str, optional] the spectral type of the star, e.g. "A0V", "G5III"

Returns

source [`simcado.Source`]

Notes

The units of the A0V spectrum in `source` are [ph/s/m2/bin]. The weight values are the scaling factors to bring a V=0 A0V spectrum down to the required magnitude for each star.

empty_sky

`simcado.source.`**`empty_sky`**`()`

Returns an empty source so that instrumental fluxes can be simulated

Returns

sky [Source]

SED

`simcado.source.`**`SED`**`(`*spec_type, filter_name='V', magnitude=0.0*`)`

Return a scaled SED for a star or type of galaxy

The SED can be for stellar spectra of galacty spectra. It is best not to mix the two types when calling `SED()`. Either provide a list of stellar types, e.g. ["G2V", "A0V"], of a list of galaxy types, e.g. ["elliptical", "starburst"]

To get the list of galaxy types that are installed, call get_SED_names(). All stellar types from the Pickles (1998) catalogue are available.

Parameters

spec_type [str, list] The spectral type of the star(s) - from the Pickles 1998 catalogue The names of a galaxy spectrum - see get_SED_names()

filter_name [str, optional] Default is "V". Any filter in the simcado/data directory can be used, or the user can specify a file path to an ASCII file for the filter

magnitude [float, list, optional] Apparent magnitude of the star. Default is 0.

Returns

lam [np.ndarray] [um] The centre of each 5 Ang bin along the spectral axis

val [np.ndarray] [ph/s/m2/bin] The photon flux of the star in each bin

Notes

Original flux units for the stellar spectra are in [ph/s/m2/AA], so we multiply the flux by 5 to get [ph/s/m2/bin]. Therefore divide by 5*1E4 if you need the flux in [ph/s/cm2/Angstrom]

Examples

Get the SED and the wavelength bins for a J=0 A0V star

```
>>> from simcado.source import SED
>>> lam, spec = SED("A0V", "J", 0)
```

Get the SED for a generic starburst galaxy

```
>>> lam, spec = SED("starburst")
```

Get the SEDs for several spectral types with different magnitudes

```
import matplotlib.pyplot as plt
from simcado.source import SED

lam, spec = SED(spec_type=["A0V", "G2V"],
                    filter_name="Pa-beta",
                    magnitude=[15, 20])

plt.plot(lam, spec[0], "blue", label="Vega")
plt.plot(lam, spec[1], "orange", label="G2V")
plt.semilogy(); plt.legend(); plt.show()
```

redshift_SED

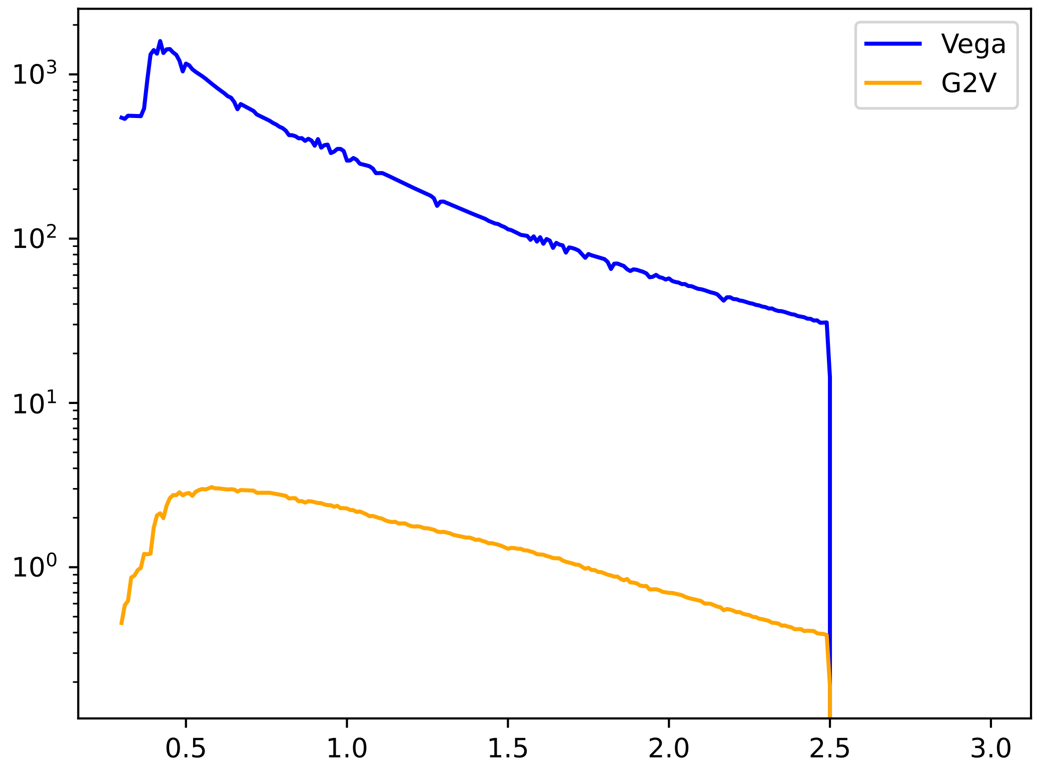
`simcado.source.`**`redshift_SED`**`(`*z*, *spectrum*, *mag*, *filter_name='TC_filter_Ks.dat'*`)`

Redshift a SimCADO SED and scale it to a magnitude in a user specified filter

Parameters

z: redshift of the source

spectrum: str, EmissionCurve, optional The spectrum to be associated with the source. Values can either be: - the name of a SimCADO SED spectrum : see get_SED_names() - an EmissionCurve with a user defined spectrum


Vega
G2V
10^3
10^2
10^1
10^0
0.5
1.0
1.5
2.0
2.5
3.0

mag: magnitude to scale the the SED after redshifting the spectrum

filter_name: filter in which the magnitude is given

Returns

ec: EmissionCurve object

See also:

`get_SED_names`

`SED`

`scale_spectrum`

Notes

wavelength and flux of the redshifted spectrum can be accessed with ec.lam and ec.val

the returned object can directly be used in any source function that accepts an EmissionCurve object (source.elliptical, source.spiral, source.point_source)

sie_grad

`simcado.source.`**`sie_grad`**(*x*, *y*, *par*)

Compute the deflection of an SIE (singular isothermal ellipsoid) potential

Parameters

x, y [meshgrid arrays] vectors or images of coordinates; should be matching numpy ndarrays

par [list] vector of parameters with 1 to 5 elements, defined as follows: par[0]: lens strength, or 'Einstein radius' par[1]: (optional) x-center (default = 0.0) par[2]: (optional) y-center (default = 0.0) par[3]: (optional) axis ratio (default=1.0) par[4]: (optional) major axis Position Angle

in degrees c.c.w. of x axis. (default = 0.0)

Returns

xg, yg [gradients at the positions (x, y)]

Notes

This routine implements an 'intermediate-axis' convention.

Analytic forms for the SIE potential can be found in: Kassiola & Kovner 1993, ApJ, 417, 450 Kormann et al. 1994, A&A, 284, 285 Keeton & Kochanek 1998, ApJ, 495, 157

The parameter-order convention in this routine differs from that of a previous IDL routine of the same name by ASB.

apply_grav_lens

`simcado.source.`**`apply_grav_lens`**(*image*, *x_cen=0*, *y_cen=0*, *r_einstein=None*, *eccentricity=1*, *rotation=0*)

Apply a singular isothermal ellipsoid (SIE) gravitational lens to an image

Parameters

image [np.ndarray]

x_cen, y_cen [float] [pixel] centre of the background image relative to the centre of the field of view

r_einstein [float] [pixel] Einstein radius of lens. If None, r_einstein = image.shape[0] // 4

eccentricity [float] [1..0] The ratio of semi-minor to semi-major axis for the lens

rotation [float] [degrees] Rotation of lens ccw from the x axis

Returns

lensed_image [np.ndarray]

get_SED_names

`simcado.source.`**`get_SED_names`**`(`*path=None*`)`

Return a list of the SEDs installed in the package directory

Looks for files that follow the naming convention `SED_<name>.dat`. For example, SimCADO contains an SED for an elliptical galaxy named `SED_elliptical.dat`

Parameters

path [str, optional] Directory to look in for filters

Returns

sed_names [list] A list of names for the SED files available

See also:

`SED()`

Examples

Names returned here can be used with the function `SED()` to call up

```
>>> from simcado.source import SED, get_SED_names
>>> print(get_SED_names())
['elliptical', 'interacting', 'spiral', 'starburst', 'ulirg']
>>> SED("spiral")
(array([ 0.3  ,  0.301,  0.302, ...,  2.997,  2.998,  2.999]),
 array([        0.        ,         0.        ,  26055075.98709349, ...,
         5007498.76444208,   5000699.21993188,   4993899.67542169]))
```

scale_spectrum

`simcado.source.`**`scale_spectrum`**`(`*lam, spec, mag, filter_name='Ks', return_ec=False*`)`

Scale a spectrum to be a certain magnitude

Parameters

lam [np.ndarray] [um] The wavelength bins for spectrum

spec [np.ndarray] The spectrum to be scaled into [ph/s/m2] for the given broadband filter

mag [float] magnitude of the source

filter_name [str, TransmissionCurve, optional] Any filter name from SimCADO or a `TransmissionCurve` object (see `get_filter_set()`)

return_ec [bool, optional] If True, a `simcado.spectral.EmissionCurve` object is returned. Default is False

Returns

lam [np.ndarray] [um] The centres of the wavelength bins for the new spectrum

spec [np.ndarray] [ph/s/m2] The spectrum scaled to the specified magnitude

If return_ec == True, a `simcado.spectral.EmissionCurve` is returned

See also:

`simcado.spectral.TransmissionCurve`

`simcado.optics.get_filter_curve()`

`simcado.optics.get_filter_set()`

`simcado.source.SED()`

`simcado.source.stars()`

Examples

Scale the spectrum of a G2V star to J=25:

```
>>> lam, spec = simcado.source.SED("G2V")
>>> lam, spec = simcado.source.scale_spectrum(lam, spec, 25, "J")
```

Scale the spectra for many stars to different H-band magnitudes:

```
>>> from simcado.source import SED, scale_spectrum
>>>
>>> star_list = ["A0V", "G2V", "M5V", "B6III", "O9I", "M2IV"]
>>> magnitudes = [ 20,   25.5,   29.1,      17,   14.3,    22    ]
>>> lam, spec = SED(star_list)
>>> lam, spec = scale_spectrum(lam, spec, magnitudes, "H")
```

Re-scale the above spectra to the same magnitudes in Pa-Beta:

```
>>> # Find which filters are in the simcado/data directory
>>>
>>> import simcado.optics as sim_op
>>> print(sim_op.get_filter_set())
['xH1', 'xY2', 'Spec_IJ', 'K-cont', 'I', 'xI2', 'Ks2', 'xY1', 'R',
'Y', 'Br-gamma', 'J-long', 'Pa-beta', 'H', 'I-long', 'H2_1-0S1',
'H-short', 'H-long', 'He-I', 'K-short', 'xJ2', 'J', 'xJ1', 'V', 'FeII',
'xI1', 'xK2', 'K-long', 'K-mid', 'J-short', 'H-cont', 'xK1', 'B', 'U',
'Ks', 'xH2', 'Spec_HK']
>>>
>>> lam, spec = scale_spectrum(lam, spec, magnitudes, "Pa-beta")
```

scale_spectrum_sb

`simcado.source.`**`scale_spectrum_sb`**(*lam*, *spec*, *mag_per_arcsec*, *pix_res=0.004*, *filter_name='Ks'*, *return_ec=False*)

Scale a spectrum to be a certain magnitude per arcsec2

Parameters

lam [np.ndarray] [um] The wavelength bins for spectrum

spec [np.ndarray] The spectrum to be scaled into [ph/s/m2] for the given broadband filter

mag_per_arcsec [float] [mag/arcsec2] surface brightness of the source

pix_res [float] [arcsec] the pixel resolution

filter_name [str, TransmissionCurve] Any filter name from SimCADO or a `TransmissionCurve` object (see `get_filter_set()`)

return_ec [bool, optional] If True, a `simcado.spectral.EmissionCurve` object is returned. Default is False

Returns

lam [np.ndarray] [um] The centres of the wavelength bins for the new spectrum

spec [np.array] [ph/s/m2/pixel] The spectrum scaled to the specified magnitude

flat_spectrum

`simcado.source.`**`flat_spectrum`**(*mag*, *filter_name='Ks'*, *return_ec=False*)

Return a flat spectrum scaled to a certain magnitude

Parameters

mag [float] [mag] magnitude of the source

filter_name [str, TransmissionCurve, optional] str - filter name. See `simcado.optics.get_filter_set()`. Default: "Ks" TransmissionCurve - output of `simcado.optics.get_filter_curve()`

return_ec [bool, optional] If True, a simcado.spectral.EmissionCurve object is returned. Default is False

Returns

lam [np.ndarray] [um] The centres of the wavelength bins for the new spectrum

spec [np.array] [ph/s/m2/arcsec] The spectrum scaled to the specified magnitude

flat_spectrum_sb

`simcado.source.`**`flat_spectrum_sb`**(*mag_per_arcsec*, *filter_name='Ks'*, *pix_res=0.004*, *return_ec=False*)

Return a flat spectrum for a certain magnitude per arcsec

Parameters

mag_per_arcsec [float] [mag/arcsec2] surface brightness of the source

filter_name [str, TransmissionCurve, optional] str - filter name. See `simcado.optics.get_filter_set()`. Default: "Ks" TransmissionCurve - output of `simcado.optics.get_filter_curve()`

pix_res [float] [arcsec] the pixel resolution. Default is 4mas (i.e. 0.004)

return_ec [bool, optional] Default is False. If True, a simcado.spectral.EmissionCurve object is returned.

Returns

lam [np.ndarray] [um] The centres of the wavelength bins for the new spectrum

spec [np.array] [ph/s/m2/arcsec] The spectrum scaled to the specified magnitude

value_at_lambda

`simcado.source.`**`value_at_lambda`**`(`*lam_i*, *lam*, *val*, *return_index=False*`)`

Return the value at a certain wavelength - i.e. val[lam] = x

Parameters

lam_i [float] the wavelength of interest

lam [np.ndarray] an array of wavelengths

val [np.ndarray] an array of values

return_index [bool, optional] If True, the index of the wavelength of interest is returned Default is False

BV_to_spec_type

`simcado.source.`**`BV_to_spec_type`**`(`*B_V*`)`

Returns the latest main sequence spectral type(s) for (a) B-V colour

Parameters

B_V [float, array] [mag] B-V colour

Returns

spec_types [list] A list of the spectral types corresponding to the B-V colours

Examples

```
>>> BV = np.arange(-0.3, 2.5, 0.5)
>>> spec_types = BV_to_spec_type(BV)
>>> print(BV)
>>> print(spec_types)
[-0.3  0.2  0.7  1.2  1.7  2.2]
['O9V', 'A8V', 'G2V', 'K5V', 'M3V', 'M8V']
```

zero_magnitude_photon_flux

`simcado.source.`**`zero_magnitude_photon_flux`**`(`*filter_name*`)`

Return the number of photons for a m=0 star for a filter with synphot

Parameters

filter_name [str]

filter name. See simcado.optics.get_filter_set()

Notes

units in [ph/s/m2]

mag_to_photons

`simcado.source.`**`mag_to_photons`**(*filter_name*, *magnitude=0*)

Return the number of photons for a certain filter and magnitude

Parameters

filter_name [str] filter name. See simcado.optics.get_filter_set()

magnitude [float] [mag] the source brightness

Returns

flux [float] [ph/s/m2] Photon flux in the given filter

See also:

`photons_to_mag()`

`zero_magnitude_photon_flux()`

`simcado.optics.get_filter_set()`

photons_to_mag

`simcado.source.`**`photons_to_mag`**(*filter_name*, *photons=1*)

Return the number of photons for a certain filter and magnitude

Parameters

filter_name [str] filter name. See simcado.optics.get_filter_set()

photons [float] [ph/s/m2] the integrated photon flux for the filter

Returns

mag [float] The magnitude of an object with the given photon flux through the filter

See also:

`photons_to_mag()`

`zero_magnitude_photon_flux()`

`simcado.optics.get_filter_set()`

_get_pickles_curve

`simcado.source._get_pickles_curve(spec_type, cat=None, verbose=False)`

Returns the emission curve for a single or list of `spec_type`, normalised to 5556A

Parameters

spec_type [str, list] The single (or list) of spectral types (i.e. "A0V" or ["K5III", "B5I"])

Returns

lam [np.array]

lam [np.array] a single np.ndarray for the wavelength bins of the spectrum,

val [np.array (list)] a (list of) np.ndarray for the emission curve of the spectral type(s) relative to the flux at 5556A

References

Pickles 1998 - DOI: 10.1086/316197

_get_stellar_properties

`simcado.source._get_stellar_properties(spec_type, cat=None, verbose=False)`

Returns an `astropy.Table` with the list of properties for the star(s) in `spec_type`

Parameters

spec_type [str, list] The single or list of spectral types

cat [str, optional] The filename of a catalogue in a format readable by `astropy.io.ascii.read()`, e.g. ASCII, CSV. The catalogue should contain stellar properties

verbose [bool] Print which stellar type is being considered

Returns

props [`astropy.Table` or list of `astropy.Table` objects] with stellar parameters

_get_stellar_Mv

`simcado.source._get_stellar_Mv(spec_type)`

Returns a single (or list of) float(s) with the V-band absolute magnitude(s)

Parameters

spec_type [str, list] The single or list of spectral types

Returns

Mv [float, list]

_get_stellar_mass

`simcado.source._get_stellar_mass(spec_type)`

Returns a single (or list of) float(s) with the stellar mass(es)

Parameters

spec_type [str, list] The single or list of spectral types in the normal format: G2V

Returns

mass [float, list] [Msol]

Classes

`Source`([filename, lam, spectra, x, y, ref, . . .])	Create a source object from a file or from arrays

Source

class `simcado.source.`**`Source`**(*filename=None*, *lam=None*, *spectra=None*, *x=None*, *y=None*, *ref=None*, *weight=None*, ***kwargs*)

Bases: `object`

Create a source object from a file or from arrays

Source class generates the arrays needed for source. It takes various inputs and converts them to an array of positions and references to spectra. It also converts spectra to photons/s/voxel. The default units for input data is ph/s/m2/bin.

The internal variables are related like so:

```
f(x[i], y[i]) = spectra[ref[i]] * weight[i]
```

Parameters

filename [str] FITS file that contains either a previously saved `Source` object or a data cube with dimensions x, y, lambda. A `Source` object is identified by the header keyword SIMCADO with value SOURCE.

or

lam [np.array] [um] Wavelength bins of length (m)

spectra [np.array] [ph/s/m2/bin] A (n, m) array with n spectra, each with m spectral bins

x, y [np.array] [arcsec] coordinates of where the emitting sources are relative to the centre of the field of view

ref [np.array] the index for .spectra which connects a position (x, y) to a spectrum f(x[i], y[i]) = spectra[ref[i]] * weight[i]

weight [np.array] A weighting to scale the relevant spectrum for each position

Attributes Summary

`info_keys`	Return keys of the *info* dict

Methods Summary

`add_background_surface_brightness`(self)	Add an EmissionCurve for the background surface brightness of the object
`apply_optical_train`(self, opt_train, detector)	Apply all effects along the optical path to the source photons
`dump`(self, filename)	Save to filename as a pickle
`image_in_range`(self, psf, lam_min, lam_max, . . .)	Find the sources that fall in the chip area and generate an image for the wavelength range [lam_min, lam_max)
`load`(filename)	Load :class:'.Source' object from filename
`on_grid`(self[, pix_res])	Return an image with the positions of all sources.
`photons_in_range`(self[, lam_min, lam_max])	Number of photons between lam_min and lam_max in units of [ph/s/m2]
`project_onto_chip`(self, image, chip)	Re-project the photons onto the same grid as the detectors use
`read`(self, filename)	Read in a previously saved `Source` FITS file
`rotate`(self, angle[, unit, use_orig_xy])	Rotates the `x` and `y` coordinates by `angle` [degrees]
`scale_spectrum`(self[, idx, mag, filter_name])	Scale a certain spectrum to a certain magnitude
`scale_with_distance`(self, distance_factor)	Scale the source for a new distance
`shift`(self[, dx, dy, use_orig_xy])	Shifts the coordinates of the source by (dx, dy) in [arcsec]
`write`(self, filename)	Write the current Source object out to a FITS file

Attributes Documentation

`info_keys`
Return keys of the *info* dict

Methods Documentation

`add_background_surface_brightness`(*self*)
Add an EmissionCurve for the background surface brightness of the object

`apply_optical_train`(*self*, *opt_train*, *detector*, *chips='all'*, *sub_pixel=False*, ***kwargs*)
Apply all effects along the optical path to the source photons

Parameters

opt_train [simcado.OpticalTrain] the object containing all information on what is to happen to the photons as they travel from the source to the detector

detector [simcado.Detector] the object representing the detector

chips [int, str, list, optional] The IDs of the chips to be readout. "all" is also acceptable

sub_pixel [bool, optional] if sub-pixel accuracy is needed, each source is shifted individually. Default is False

Other Parameters

INST_DEROT_PERFORMANCE [float] [0 .. 100] Percentage of the sky rotation that the derotator removes

SCOPE_JITTER_FWHM [float] [arcsec] The FWMH of the gaussian blur caused by jitter

SCOPE_DRIFT_DISTANCE [float] [arcsec] How far from the centre of the field of view has the telescope drifted during a DIT

Notes

Output array is in units of [ph/s/pixel] where the pixel is internal oversampled pixels - not the pixel size of the detector chips

`dump` (*self*, *filename*)
Save to filename as a pickle

`image_in_range` (*self*, *psf*, *lam_min*, *lam_max*, *chip*, ***kwargs*)
Find the sources that fall in the chip area and generate an image for the wavelength range [lam_min, lam_max)

Output is in [ph/s/pixel]

Parameters

psf [psf.PSF object] The PSF that the sources will be convolved with

lam_min, lam_max [float] [um] the wavelength range relevant for the psf

chip [str, detector.Chip]

- detector.Chip : the chip that will be seeing this image.
- str : [“tiny”, “small”, “center”] -> [128, 1024, 4096] pixel chips

Returns

slice_array [np.ndarray] the image of the source convolved with the PSF for the given range

`classmethod load` (*filename*)
Load :class:’.Source’ object from filename

`on_grid` (*self*, *pix_res=0.004*)
Return an image with the positions of all sources.

The pixel values correspond to the number of emitting objects in that pixel

Parameters

pix_res [float] [arcsec] The grid spacing

Returns

im [2D array] A numpy array containing an image of where the sources are

`photons_in_range` (*self*, *lam_min=None*, *lam_max=None*)
Number of photons between lam_min and lam_max in units of [ph/s/m2]

Calculate how many photons for each source exist in the wavelength range defined by lam_min and lam_max.

Parameters

lam_min, lam_max [float, optional] [um] integrate photons between these two limits. If both are `None`, limits are set at lam[0], lam[-1] for the source’s wavelength range

Returns

slice_photons [float] [ph/s/m2] The number of photons in the wavelength range

`project_onto_chip` (*self*, *image*, *chip*)
Re-project the photons onto the same grid as the detectors use

Parameters

image [np.ndarray] the image to be re-projected

chip [detector.Chip] the chip object where the image will land

read (*self*, *filename*)

Read in a previously saved `Source` FITS file

Parameters

filename [str] Path to the file

rotate (*self*, *angle*, *unit='degree'*, *use_orig_xy=False*)

Rotates the `x` and `y` coordinates by `angle` [degrees]

Parameters

angle [float] Default is in degrees, this can set with `unit`

unit [str, astropy.Unit] Either a string with the unit name, or an `astropy.unit.Unit` object

use_orig_xy [bool] If the rotation should be based on the original coordinates or the current coordinates (e.g. if rotation has already been applied)

scale_spectrum (*self*, *idx=0*, *mag=20*, *filter_name='Ks'*)

Scale a certain spectrum to a certain magnitude

See `simcado.source.scale_spectrum()` for examples

Parameters

idx [int] The index of the spectrum to be scaled: <Source>.spectra[idx] Default is <Source>.spectra[0]

mag [float] [mag] new magnitude of spectrum

filter_name [str, TransmissionCurve] Any filter name from SimCADO or a `TransmissionCurve` object (see `get_filter_set()`)

scale_with_distance (*self*, *distance_factor*)

Scale the source for a new distance

Scales the positions and brightnesses of the `Source` object according to the ratio of the new and old distances

i.e. distance_factor = new_distance / current_distance

Warning: This does not yet take into account redshift

Todo: Implement redshift

Parameters

distance_factor [float] The ratio of the new distance to the current distance i.e. distance_factor = new_distance / current_distance

Examples

```
>>> from simcado.source import cluster
>>>
>>> curr_dist = 50000  # pc, i.e. LMC
>>> new_dist = 770000  # pc, i.e. M31
>>> src = cluster(distance=curr_dist)
>>> src.scale_with_distance( new_dist/curr_dist )
```

shift(*self*, *dx=0*, *dy=0*, *use_orig_xy=False*)
Shifts the coordinates of the source by (dx, dy) in [arcsec]

Parameters

dx, dy [float, array] [arcsec] The offsets for each coordinate in the arrays `x`, `y`. - If dx, dy are floats, the same offset is applied to all coordinates - If dx, dy are arrays, they must be the same length as `x`, `y`

use_orig_xy [bool] If the shift should be based on the original coordinates or the current coordinates (e.g. if shift has already been applied)

write(*self*, *filename*)
Write the current Source object out to a FITS file

Parameters

filename [str] where to save the FITS file

Notes

Just a place holder so that I know what's going on with the input table * The first extension [0] contains an "image" of size 4 x N where N is the number of sources. The 4 columns are x, y, ref, weight. * The second extension [1] contains an "image" with the spectra of all sources. The image is M x len(spectrum), where M is the number of unique spectra in the source list. M = max(ref) - 1

5.13.8 simcado.spatial module

simcado.spatial Module

TODO: Insert docstring

Functions

`tracking`(arr, cmds)	A method to simulate tracking errors ===== Currently a place holder with minimum functionality ========= !! TODO, work out the shift during the DIT for the object RA, DEC etc !!
`derotator`(arr, cmds)	A method to simulate field rotation in case the derotator is <100% effective ===== Currently a place holder with minimum functionality ========= !! TODO, work out the rotation during the DIT for the object RA, DEC etc !!

Continued on next page

Table 23 – continued from previous page

`wind_jitter`(arr, cmds)	A method to simulate wind jitter ===== Currently a place holder with minimum functionality ========= !! TODO, get the read spectrum for wind jitter !! !! Add in an angle parameter for the ellipse !!
`adc_shift`(cmds)	Generates a list of x and y shifts from a commands object
`make_distortion_maps`(real_xy, detector_xy[, ...])	Generate distortion maps based on star positions.
`get_distorion_offsets`(x, y, dist_map_hdus, ...)	Returns the distortion offsets for position relative to the FoV centre

tracking

`simcado.spatial.`**`tracking`**(*arr*, *cmds*)

A method to simulate tracking errors ===== Currently a place holder with minimum functionality ========= !! TODO, work out the shift during the DIT for the object RA, DEC etc !!

derotator

`simcado.spatial.`**`derotator`**(*arr*, *cmds*)

A method to simulate field rotation in case the derotator is <100% effective ===== Currently a place holder with minimum functionality ========= !! TODO, work out the rotation during the DIT for the object RA, DEC etc !!

wind_jitter

`simcado.spatial.`**`wind_jitter`**(*arr*, *cmds*)

A method to simulate wind jitter ===== Currently a place holder with minimum functionality ========= !! TODO, get the read spectrum for wind jitter !! !! Add in an angle parameter for the ellipse !!

adc_shift

`simcado.spatial.`**`adc_shift`**(*cmds*)

Generates a list of x and y shifts from a commands object

make_distortion_maps

`simcado.spatial.`**`make_distortion_maps`**(*real_xy*, *detector_xy*, *step=1*)

Generate distortion maps based on star positions.

The centres of the returned images correspond to the centre of the detector plane

Parameters

real_xy [list] [arcsec] Contains 2 arrays: ([x_pos], [y_pos]) where x_pos, y_pos are the coordinates of the real position of the stars

detector_xy [list] [arcsec] Contains 2 arrays: ([x_pos], [y_pos]) where x_pos, y_pos are the coordinates of the detected position of the stars

step [float] [arcsec] the grid spacing of the returned images

Returns

dx, dy [2D array] Returns two arrays with

get_distorion_offsets

`simcado.spatial.`**`get_distorion_offsets`**(*x*, *y*, *dist_map_hdus*, *corners*)

Returns the distortion offsets for position relative to the FoV centre

Parameters

x, y [float, list] [arcsec] Distances from the centre of the distortion map (which should also be the centre of the FoV, given by CPREFn header Keywords)

dist_map_hdus [list, astropy.io.fits.HDUList] The distortion maps in X and Y directions. Accepts either - a list of two PrimaryHDU objects, each containing a map of the

distortion on the x and y direction. E.g. `dist_map_hdus=(hdu_dx, hdu_dy)` where `hdu_dx` and `hdu_dy` are FITS image objects (`ImageHDU`), or

- a HDUList object which contains 3 HDU objects: A PrimaryHDU and two ImageHDUs. - extension [0] (the PrimaryHDU) is nothing by a header, - extension [1] contains a map of the x-axis distortion, - extension [2] contains a map of the y-axis distortions

corners [list] [arcsec] A list containing 4 values for the borders of the distortion map grid in the following order (x_min, x_max, y_min, y_max)

Returns

dx, dy [float, list] [arcsec] the shifts which need to be applied to the input positions x,y

See also:

`make_distortion_maps`

`astropy.io.fits.HDUList`

`astropy.io.fits.PrimaryHDU`

5.13.9 `simcado.spectral module`

simcado.spectral Module

Classes for spectral curves

Functions

`get_sky_spectrum`(fname, airmass[, return_type])	Return a spectral curve for the sky for a certain airmass

get_sky_spectrum

`simcado.spectral.`**`get_sky_spectrum`**(*fname*, *airmass*, *return_type=None*, ***kwargs*)

Return a spectral curve for the sky for a certain airmass

Parameters

fname [str] the file containing the spectral curves

airmass [float, optinal] Default is 1.0. Acceptable values are between 1.0 and 3.0

return_type [str, optional] [“transmission”, “emission”, None] Default is None. A TransmissionCurve or EmissionCurve object will be returned if desired. If None two arrays are returned: (lam, val)

****kwargs** [optional] kwargs are passed directly onto the TransmissionCurve or EmissionCurve classes

Returns

TransmissionCurve or EmissionCurve or (lam, val) By default lam is in [um] and val [ph/s/m2/um/arcsec2] if val is an emission spectrum

Notes

This function is designed to work with a table of values produced by SkyCalc The column names must begin with lambda and then columns must be named according to the airmass following this pattern: “X1.5” for an airmass of 1.5

Classes

`TransmissionCurve`([filename, lam, val])	Very basic class to either read in a text file for a transmission curve or take two vectors to make a transmission curve
`EmissionCurve`([filename])	Class for emission curves
`BlackbodyCurve`(lam, temp, **kwargs)	Blackbody emission curve
`UnityCurve`([lam, val])	Constant transmission curve

TransmissionCurve

class `simcado.spectral.`**`TransmissionCurve`**(*filename=None*, *lam=None*, *val=None*, ***kwargs*)

Bases: `object`

Very basic class to either read in a text file for a transmission curve or take two vectors to make a transmission curve

Parameters

filename [str, optional] The path to the file containing wavelength and transmission data where the first column is wavelength in [um] and the second is the transmission coefficient between [0,1].

Alternatively this data can be passed directly. If filename is not provided, `lam=` and `val=` must be passed

lam [array, optional] [um] Wavelength bins in a 1D numpy array of length n

val [array, optional] [0 .. 1] The transmission coefficients

Returns

TransmissionCurve object

Methods Summary

`filter_info`(self)	Returns the filter properties as a dictionary
`filter_table`(self)	Returns the filter properties as a astropy.table
`normalize`(self[, val, mode])	Normalize the spectral curve
`plot`(self, **kwargs)	Plot the transmission curve on the current axis
`resample`(self, bins[, action, use_edges, . . .])	Resamples both the wavelength and value vectors to an even grid.

Methods Documentation

`filter_info`(*self*)
Returns the filter properties as a dictionary

`filter_table`(*self*)
Returns the filter properties as a astropy.table

Notes

ONLY works if filter files have the SimCADO header format

The following keywords should be in the header:

```
author
source
date_created
date_modified
status
type
center
width
blue_cutoff
red_cutoff
```

`normalize`(*self*, *val=1.0*, *mode='integral'*)
Normalize the spectral curve

Parameters

val [float, optional] The value to normalise to. Default is 1.

mode [str, optional]

- **"integral" normalizes the integral over the defined** wavelength range to val (default: 1.)
- **"maximum" normalizes the maximum over the defined** wavelength range to val (default: 1.)

`plot`(*self*, ***kwargs*)
Plot the transmission curve on the current axis

The method accepts matplotlib.pyplot keywords.

`resample`(*self*, *bins*, *action='average'*, *use_edges=False*, *min_step=None*, *use_default_lam=False*)
Resamples both the wavelength and value vectors to an even grid. In order to avoid losing spectral information, the TransmissionCurve resamples down to a resolution of 'min_step' (default: 0.01nm) before resampling again up to the given sampling vector defined by 'bins'.

Parameters

bins [float or array of floats]

[um]: float - taken to mean the width of bins on an even grid

array - the centres of the spectral bins

- the edges of the spectral bins if use_edges = True

action [str, optional] ['average','sum'] How to rebin the spectral curve. If 'sum', then the curve is normalised against the integrated value of the original curve. If 'average', the average value per bin becomes the value for each bin.

use_edges [bool, optional] [False, True] True if the array passed in 'bins' describes the edges of the wavelength bins. Default is False

min_step [float, optional] [um] default=1E-4, the step size for the down-sample

use_default_lam [bool, optional] Default is False. If True, `bins` is ignored and the default wavelength range is used as the resampling grid.

EmissionCurve

class `simcado.spectral.`**`EmissionCurve`**(*filename=None*, ***kwargs*)

Bases: `simcado.spectral.TransmissionCurve`

Class for emission curves

Create an emission curve from a file or from wavelength and flux vectors. In the latter case, the keywords *lam* and *val* have to be specified.

Parameters

- filename: string with the path to the transmission curve file where the first column is wavelength in [um] and the second is the transmission coefficient between [0,1]

- lam: [um] 1D numpy array of length n

- val: 1D numpy array of length n

- res: [um] float with the desired spectral resolution

- pix_res: [arcsec] float of int for the field of view for each pixel

- area: [m2] float or int for the collecting area of M1

- units: string or astropy.unit for calculating the number of photons per voxel

Return values are in [ph/s/voxel]

Examples

```
>>> from simcado.spectral import EmissionCurve
>>>
>>> ec_1 = EmissionCurve("emission_curve.dat")
>>> lam = np.arange(0.7, 1.5, 0.05)
>>> flux = 1. - 0.2 * wave**2   # power-law spectrum
>>> ec_2 = EmissionCurve(lam=lam, val=flux)
```

Methods Summary

`convert_to_photons`(self)	Do the conversion to photons/s/voxel by using the val_unit, lam, area and exptime keywords.
`photons_in_range`(self[, lam_min, lam_max])	Sum up the photons in the wavelength range [lam_min, lam_max]
`resample`(self, bins[, action, use_edges, . . .])	Rebin an emission curve

Methods Documentation

`convert_to_photons`(*self*)

Do the conversion to photons/s/voxel by using the val_unit, lam, area and exptime keywords. If not given, make some assumptions.

`photons_in_range`(*self*, *lam_min=None*, *lam_max=None*)

Sum up the photons in the wavelength range [lam_min, lam_max]

Parameters

- lam_min, lam_max: the wavelength limits

Return values are in [ph/s/pixel]

`resample`(*self*, *bins*, *action='average'*, *use_edges=False*, *min_step=None*, *use_default_lam=False*)

Rebin an emission curve

BlackbodyCurve

class `simcado.spectral.`**`BlackbodyCurve`**(*lam*, *temp*, ***kwargs*)

Bases: `simcado.spectral.EmissionCurve`

Blackbody emission curve

Parameters

lam [1D np.ndarray] [um] the centres of the wavelength bins

temp [float] [deg C] float for the average temperature of the blackbody

pix_res [float, optional] [arcsec] Default is 0.004. Field of view for each pixel

area [float, optional] [m2] Default is 978m2. Area of the emitting surface

Returns

EmissionCurve object with units of [ph/s/voxel], i.e. photons per second per wavelength bin per full area per pixel field of view

UnityCurve

class `simcado.spectral.`**`UnityCurve`**(*lam=array([0.3, 3.])*, *val=1*, ***kwargs*)

Bases: `simcado.spectral.TransmissionCurve`

Constant transmission curve

Parameters

- lam [um]: wavelength array

- val: constant value of transmission (default: 1)

5.13.10 `simcado.utils` module

simcado.utils Module

Helper functions for SimCADO

Functions

`add_SED_to_simcado`(file_in[, file_out, …])	Adds the SED given in `file_in` to the SimCADO data directory
`add_keyword`(filename, keyword, value[, …])	Add a keyword, value pair to an extension header in a FITS file
`add_mags`(mags)	Returns a combined magnitude for a group of objects with `mags`
`airmass2zendist`(airmass)	Convert airmass to zenith distance
`airmass_to_zenith_dist`(airmass)	returns zenith distance in degrees
`angle_in_arcseconds`(distance, width)	Returns the angular distance of an object in arcseconds.
`atmospheric_refraction`(lam[, z0, temp, …])	Compute atmospheric refraction
`bug_report`()	Get versions of dependencies for inclusion in bug report
`deriv_polynomial2d`(poly)	Derivatives (gradient) of a Polynomial2D model
`dist_mod_from_distance`(d)	mu = 5 * np.log10(d) - 5
`distance_from_dist_mod`(mu)	d = 10**(1 + mu / 5)
`download_file`(url[, save_dir])	Download the extra data that aren't in the SimCADO package
`find_file`(filename[, path, silent])	Find a file in search path
`get_extras`()	Downloads large files that SimCADO needs to simulate MICADO
`is_fits`(filename)	Checks if file is a FITS file based on extension
`moffat`(r, alpha, beta)	!!Unfinished!! Return a Moffat function
`msg`(cmds, message[, level])	Prints a message based on the level of verbosity given in cmds
`nearest`(arr, val)	Return the index of the value from 'arr' which is closest to 'val'
`parallactic_angle`(ha, de[, lat])	Compute the parallactic angle
`poissonify`(arr)	Add a realisation of the poisson process to the array 'arr'.
`quantify`(item, unit)	Ensure an item is a Quantity
`seq`(start, stop[, step])	Replacement for numpy.arange modelled after R's seq function
`telescope_diffraction_limit`(aperture_size, …)	Returns the diffraction limit of a telescope
`transverse_distance`(angle, distance)	Turn an angular distance into a proper transverse distance
`unify`(x, unit[, length])	Convert all types of input to an astropy array/unit pair
`zendist2airmass`(zendist)	Convert zenith distance to airmass
`zenith_dist_to_airmass`(zenith_dist)	`zenith_dist` is in degrees

add_SED_to_simcado

`simcado.utils.`**`add_SED_to_simcado`**(*file_in*, *file_out=None*, *lam_units='um'*)

Adds the SED given in `file_in` to the SimCADO data directory

Parameters

file_in [str] path to the SED file. Can be either FITS or ASCII format with 2 columns Column 1 is the wavelength, column 2 is the flux

file_out [str, optional] Default is None. The file path to save the ASCII file. If `None`, the SED is saved to the SimCADO data directory i.e. to `<utils.__pkg_dir__>/data/`

lam_units [str, astropy.Units] Units for the wavelength column, either as a string or as astropy units Default is [um]

add_keyword

`simcado.utils.`**`add_keyword`**(*filename*, *keyword*, *value*, *comment=''*, *ext=0*)

Add a keyword, value pair to an extension header in a FITS file

Parameters

filename [str] Name of the FITS file to add the keyword to

keyword [str]

value [str, float, int]

comment [str]

ext [int, optional] The fits extension index where the keyword should be added. Default is 0

add_mags

`simcado.utils.`**`add_mags`**(*mags*)

Returns a combined magnitude for a group of objects with `mags`

airmass2zendist

`simcado.utils.`**`airmass2zendist`**(*airmass*)

Convert airmass to zenith distance

Parameters

airmass [float (>= 1)]

Returns

zenith distance in degrees

airmass_to_zenith_dist

`simcado.utils.`**`airmass_to_zenith_dist`**(*airmass*)

returns zenith distance in degrees

Z = arccos(1/X)

angle_in_arcseconds

`simcado.utils.`**`angle_in_arcseconds`**`(`*distance*, *width*`)`

Returns the angular distance of an object in arcseconds. Units must be consistent

atmospheric_refraction

`simcado.utils.`**`atmospheric_refraction`**`(`*lam*, *z0=60*, *temp=0*, *rel_hum=60*, *pres=750*, *lat=-24.5*, *h=3064*`)`

Compute atmospheric refraction

The function computes the angular difference between the apparent position of a star seen from the ground and its true position.

Parameters

lam [float, np.ndarray] [um] wavelength bin centres

z0 [float, optional] [deg] zenith distance. Default is 60 deg from zenith

temp [float, optional] [deg C] ground temperature. Default is 0 deg C

rel_hum [float, optional] [%] relative humidity. Default is 60%

pres [float, optional] [millibar] air pressure. Default is 750 mbar

lat [float, optional] [deg] latitude. Default set for Cerro Armazones: 24.5 deg South

h [float, optional] [m] height above sea level. Default is 3064 m

Returns

ang [float, np.ndarray] [arcsec] angle between real position and refracted position

References

See Stone 1996 and the review by S. Pedraz - http://www.caha.es/newsletter/news03b/pedraz/newslet.html

bug_report

`simcado.utils.`**`bug_report`**`()`

Get versions of dependencies for inclusion in bug report

deriv_polynomial2d

`simcado.utils.`**`deriv_polynomial2d`**`(`*poly*`)`

Derivatives (gradient) of a Polynomial2D model

Parameters

poly [astropy.modeling.models.Polynomial2D]

dist_mod_from_distance

`simcado.utils.`**`dist_mod_from_distance`**`(`*d*`)`

mu = 5 * np.log10(d) - 5

distance_from_dist_mod

`simcado.utils.`**`distance_from_dist_mod`**(*mu*)

d = 10**(1 + mu / 5)

download_file

`simcado.utils.`**`download_file`**(*url*, *save_dir='/home/docs/checkouts/readthedocs.org/user_builds/simcado/checkouts/latest/s*

Download the extra data that aren't in the SimCADO package

find_file

`simcado.utils.`**`find_file`**(*filename*, *path=None*, *silent=False*)

Find a file in search path

Parameters

filename [str] name of a file to look for

path [list] list of directories to search (default: ['./'])

silent [bool] if True, remain silent when file is not found

Returns

Absolute path of the file

get_extras

`simcado.utils.`**`get_extras`**()

Downloads large files that SimCADO needs to simulate MICADO

is_fits

`simcado.utils.`**`is_fits`**(*filename*)

Checks if file is a FITS file based on extension

Parameters

filename [str]

Returns

flag [bool]

moffat

`simcado.utils.`**`moffat`**(*r*, *alpha*, *beta*)

!!Unfinished!! Return a Moffat function

Parameters

r

alpha

beta

Returns

eta

msg

`simcado.utils.`**`msg`**(*cmds*, *message*, *level=3*)

Prints a message based on the level of verbosity given in cmds

Parameters

cmds [UserCommands] just for the SIM_VERBOSE and SIM_MESSAGE_LEVEL keywords

message [str] message to be printed

level [int, optional] all messages with level <= SIM_MESSAGE_LEVEL are printed. I.e. level=5 messages are not important, level=1 are very important

nearest

`simcado.utils.`**`nearest`**(*arr*, *val*)

Return the index of the value from 'arr' which is closest to 'val'

Parameters

arr [np.ndarray, list, tuple] Array to be searched

val [float, int] Value to find in `arr`

Returns

i [int] index of array where the nearest value to `val` is

parallactic_angle

`simcado.utils.`**`parallactic_angle`**(*ha*, *de*, *lat=-24.589167*)

Compute the parallactic angle

Parameters

ha [float] [hours] hour angle of target point

de [float] [deg] declination of target point

lat [float] [deg] latitude of observatory, defaults to Armazones

Returns

parang [float] The parallactic angle

Notes

The parallactic angle is defined as the angle PTZ, where P is the .. math:: taneta = frac{cosphisin H}{sinphi cosdelta - cosphi sindelta cos H} It is negative (positive) if the target point is east (west) of the meridian.

References

R. Ball: "A Treatise on Spherical Astronomy", Cambridge 1908

poissonify

`simcado.utils.`**`poissonify`**`(`*arr*`)`

Add a realisation of the poisson process to the array 'arr'.

Parameters

arr [np.ndarray] The input array which needs a Poisson distribution applied to items

Returns

arr [np.ndarray] The input array, but with every pixel altered according to a poisson distribution

quantify

`simcado.utils.`**`quantify`**`(`*item*, *unit*`)`

Ensure an item is a Quantity

Parameters

item [int, float, array, list, Quantity]

unit [str, Unit]

Returns

quant [Quantity]

seq

`simcado.utils.`**`seq`**`(`*start*, *stop*, *step=1*`)`

Replacement for numpy.arange modelled after R's seq function

Returns an evenly spaced sequence from start to stop. stop is included if the difference between start and stop is an integer multiple of step.

From the documentation of numpy.range: "When using a non-integer step, such as 0.1, the results will often not be consistent." This replacement aims to avoid these inconsistencies.

Parameters

start, stop: [int, float] the starting and (maximal) end values of the sequence.

step [[int, float]] increment of the sequence, defaults to 1

telescope_diffraction_limit

`simcado.utils.`**`telescope_diffraction_limit`**`(`*aperture_size*, *wavelength*, *distance=None*`)`

Returns the diffraction limit of a telescope

Parameters

aperture_size [float] [m] The diameter of the primary mirror

wavelength [float] [um] The wavelength for diffarction

distance [float, optional] Default is None. If `distance` is given, the transverse distance for the diffraction limit is returned in the same units as `distance`

Returns

diff_limit [float] [arcsec] The angular diffraction limit. If distance is not None, diff_limit is in the same units as distance

transverse_distance

`simcado.utils.`**`transverse_distance`**(*angle*, *distance*)

Turn an angular distance into a proper transverse distance

Parameters

angle [float] [arcsec] The on-sky angle

distance [float] The distance to the object. Units are arbitary

Returns

trans_distance [float] proper transverse distance. Has the same Units as `distance`

unify

`simcado.utils.`**`unify`**(*x*, *unit*, *length=1*)

Convert all types of input to an astropy array/unit pair

Parameters

x [int, float, np.ndarray, astropy.Quantity] The array to be turned into an astropy.Quantity

unit [astropy.Quantity] The units to attach to the array

length [int, optional] If `x` is a scalar, and the desired output is an array with `length`

Returns

y [astropy.Quantity]

zendist2airmass

`simcado.utils.`**`zendist2airmass`**(*zendist*)

Convert zenith distance to airmass

Parameters

zenith distance [[deg]] Zenith distance angle

Returns

airmass in sec(z) approximation

zenith_dist_to_airmass

`simcado.utils.`**`zenith_dist_to_airmass`**(*zenith_dist*)

`zenith_dist` is in degrees

X = sec(Z)

CHAPTER

SIX

BUGS AND ISSUES

We freely admit that there may still be several bugs that we have not found. If you come across an buggy part of SimCADO, *please please* tell us. We can't make SimCADO better if we don't know about things.

The preferable option is to open an issue on our Github page: astronomyk/SimCADO/issues, or you can contact one of us directly.

CHAPTER

SEVEN

CONTACT

For questions and complaints alike, please contact the authors:

- kieran.leschinski@univie.ac.at
- oliver.czoske@univie.ac.at
- miguel.verdugo@unive.ac.at

Developers (Vienna): Kieran Leschinski, Oliver Czoske, Miguel Verdugo

Data Flow Team Leader (Gronigen): Gijs Verdoes Kleijn

MICADO home office (MPE): http://www.mpe.mpg.de/ir/micado

PYTHON MODULE INDEX

INDEX